ENHANCEMENTS TO IITKIX: DATA SHARING AND SEMAPHORES

by

G. RAMAMOORTHY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JANUARY, 1988

ENHANCEMENTS TO IITKIX: DATA SHARING AND SEMAPHORES

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

G. RAMAMOORTHY

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JANUARY, 1988

CERTIFICATE

This is to certify that the thesis work entitled **Enhancements to IITKIX: DATA SHARING AND SEMAPHORES** has been carried out under my supervision and has not been submitted elsewhere for the award of a similar degree.

Dr.Gautam Barua,
Asst. Professor,
Dept. of C.S.E.,
IIT, Kanpur.
January 1988.

ACKNOWLEDGEMENTS

I wish to express my heartfelt thanks to my guide, Dr.Gautam Barua , for his inspiring and patient guidance throughout the project.

I would like to thank Mr. Vijay K. Aggarwal, for his company and help during the course of my project.

My thanks are also due to all my friends, especially to the members of *A-Block Discussion Club* of Hall IV, who made my stay at IIT, Kanpur enjoyable.

Place: IIT, Kanpur.

Date : 13th January 1988.

G. Ramamoorthy

(G. RAMAMOORTHY)

ABSTRACT

This report is concerned with the enhancements made to IITKIX,a Unix System II compatiable system on the Uptron S-32. We have introduced the data sharing feature, by which many processes can share a data segment and semaphores, a synchronization tool which can be used to control access to shared resources. We have presented the implementation details, pointing out the merits and demerits of this implementation with respect to some of the existing implementations.Possible future expansions have also been suggested.

CONTENTS

1. INTRODUCTION

1.1. Historical Overview of Unix:

Unix was developed at Bell Labs as a private research project for PDP-7 computer by Ken Thompson. Later Dennis Ritchie joined with him and they announced "UNIX" to the computing world in 1974 ([RIT74]). The most attractive feature of Unix is its *portability* , which made Unix to gain such a wide popularity. Lot of enhanced versions started coming out soon. A good number of reports on porting Unix to different hardwares have been published ([COO85]).

The first version available outside the Bell Labs was Unix version 6.Ø, which ran on a PDP-11 computer. Version 7.Ø and 8.Ø were released after some time. After releasing version 7.Ø Bell Labs gave the distribution rights to Unix Software Group, which announced Unix System III and Unix System V in succession, and University of California at Berkley, which announced BSD series. Recent versions to come out of Bell Labs is System V Release 3.Ø and of UCB is 4.3 BSD. An overview of the evolution of Unix has been reported in ([QUA85]).

1.2. IITKIX

Unix Version 6 was ported to the Uptron S-32 machine and enchanced to system III. This ported version is called IITKIX. The edited code of version 6 is available in

[LIO77a]. A commentary on the same is available in [LIO77b]. The whole porting excercise has been reported in two parts ([SRI86],[KAL86]).

1.3.Aim of the project:

The aim of this project is to enhance IITKIX, by adding two new features, namely **DATA SHARING** and **SEMAPHORES.**

1.3.1.Data Sharing:

Executable files on Unix system traditionally have held all the code and initialized data for process images. One layer of sharing exists in the Unix system: processes created from the same executable file may share one copy of program text in memory. But, there is no way by which many processes can use the same copy of the data. In typical database applications, where there is a need to have a single copy of the data for consistency reasons, it is a must that all processes referring to this data must use the *same* copy. The Data Base Systems which are now available in Unix Systems (e.g . UNIFY) are implemented with the help of the Unix Kernel. They use the "BUFFER CACHE" which is used for Block IO (see chapter 2.) in Unix to hold the shared data and "locking" is implemented inside the kernel to assure consistency. Hence it is evident that we need a way by which user processes can share data without the intervention of the Operating System. This can be accomplished by mapping the virtual address spaces of different processes to the

same physical memory. Users have to ensure consistency by provididng explicit synchronization mechanisms (e.g. Semaphores).

1.3.2.Semaphores:

Semaphores allow processes to synchronize execution . Before the implementation of semaphores, a process would create a lock file with the 'creat' system call without write permission for others(or with 'open' system call with O_CREAT and O_EXCL flags set) if it wanted to lock a resource: The 'creat' fails if the file already exists and the process does not have write permission and thus the process would assume that some other process is using the resource. Later the process which has acquired the shared resource will use 'unlink' system call to delete the file, so that other process can acquire the resource.Thus this provides a binary semaphore with busy waiting. The major disadvantages of this approach are that the process does not know when to try again, and lock files may indavertently be left behind when the system crashes.

"Semaphores" are the oldest and possibly- the simplest-synchronization mechanism available. A semaphore is a non-negative integer variable on which two operation, namely **P** (decrement the value of the semaphore if its value is greater than zero or else wait) and **V** (Increment the semaphore value) , can be performed.

The semaphores can be effectively used under two distinct circumstances.

(a) **Synchronization:** When processes need to synchronize ; (i.e) First process has to proceed only after the completion of certain event , which is caused by the second process.

(b) **Mutual Exclusion:** When a particular resource has to be used by only one process; to protect the resource being accessed against other processes wishing to access it at the same time.

1.4.Organization of the thesis:

In chapter 2, an Overview of Unix Kernel has been presented with particular reference to related data structures.

Chapter 3, discusses the memory management unit of Uptron S-32, which is a major hardware aspect related with this project.

Chapter 4 , deals with the issues and implementation of data sharing, pointing out the different schemes employed by other operating systems along with some examples.

Chapter 5, is concerned with the details of semaphore implementation. An overview, along with merits and demerits of different sysnchronization mechanisms, has been presented.

Chapter 6, concludes this report by discussing some of

the important aspects of the present work. Some of the possible future expansions of our implementation are also pointed out.

2. AN OVERVIEW OF UNIX

The Unix operating system can be dividied into four parts, each concerned with one of the folllowing functions:

(1) Process Management

(2) Exception processing and system calls

(3) File system

(4) Input/Output

By Operating system here, we mean the code which is permanently resident in the core memory during the operation of UNIX, called the *Kernel.* The remaining parts of the UNIX consists of a set of programs called utilities, which run as "user programs". Under this heading come the programs such as C-shell, chmod,du etc. In this chapter we are concerned with the functions performed by the Kernel .

2.1.Process Management:

A process may be defined as a program in execution.In the Unix system, a user executes programs in an environment called *user process.* When a system function is required , the user process calls the system as a subroutine. At some point in this call there is a distinct switch of environments. After this, the process is said to be a *system process.* The user and system processes are different phases of the same process and they never execute simultaneously.

"PROC" TABLE: Every process in the system essentially contains an entry in this permanently memory resident global table. This entry contains all the data needed by the system when the process is not *active.* Examples are the process's name, the location of other segments (e.g. text segment) and scheduling information. An entry in this table is allocated when the process is created and freed when the process terminates. This process entry is always accessible by the kernel.

Each user process has its own virtual space. The core image of a user process can be divided into three segments:

1. A sharable **text segment** (read only) which begins at loction zero in the virtual space.

2. A non shared writable **data segment** which can be subdivided in to initialized and uninitialized data portion or bss.

3. Starting at the highest address in the virtual address space is a **stack segment** which grows downwards (i.e. towards lower memory addresses).

In addition, for each process there is a per process data area, called *u area,* which contains a copy of the user structure ("u" structure) and also the kernel stack for that process.

"U" STRUCTURE: The user structure contains data about the process which are required only when the process is

active. Examples of this kind of data contained in the "u" structure are : saved central processor registers, open file descriptors, accounting information and a pointer to in-core "proc" table entry corresponding to this segment.

The "u" area lies in the kernel address space and switching from one process to other is achieved simply by changing the page table entry corresponding to this page. The OS before launching a user process sets its page and segment tale entries appropriately so that one user process cannot access other process' area. A user process can dynamically expand or contract its data segment through the system call *brk*.

TEXT TABLE: Every sharable text segment has got an entry in this global system table. An entry in this table contains information like location of the segment in the secondary memory, if segment is loaded its core address, reference counts to indicate the total number of processes and number of in-core processes sharing the text.

The relationship between these segments are shown in fig.1.

A process may create another process by doing a *fork* which creats an exact copy of the process doing the fork. Another system call *exec* can be used to replace the core image of a process and start executing the new core image. A process might destroy another process by *kill* or itself by

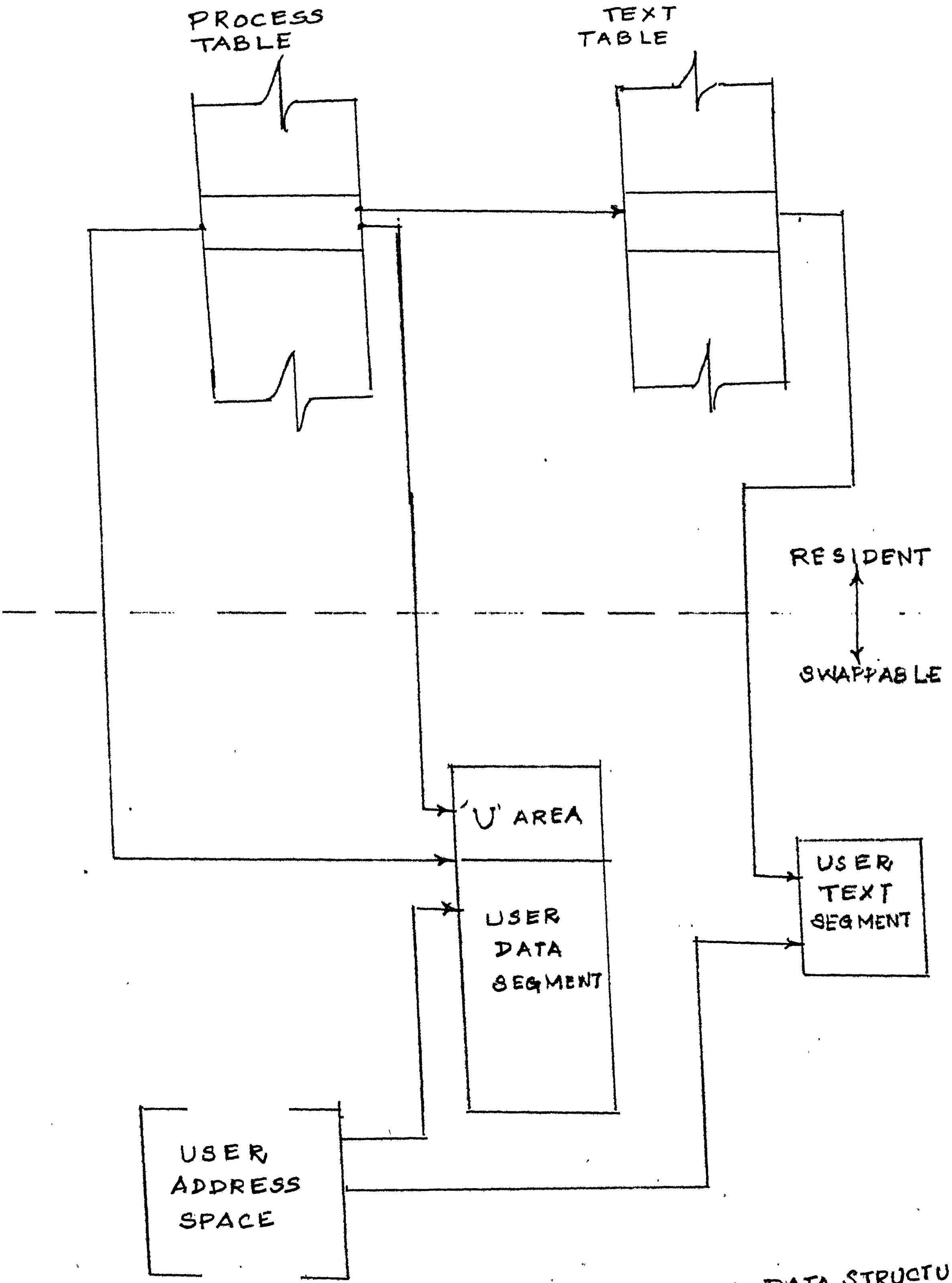

FIG.1. PROCESS CONTROL DATA STRUCTURE

exit. The *kill* system call sends a signal to the effected process. The effected process might arm itself against some signals by doing a *signal* system call.

Process synchronization is achieved by having processes sleep and wakeup on some event. An event is represented by a unique address. Every process has got a priority number which is kept in the "proc" structure. The CPU scheduler picks up from a bunch of competing processes the process whose priority is the highest. The priorities for processes are computed dynamically and depend on factors like cpu usage and the event it is waiting for.

A special process called *swapper* is always in the core and is never swapped out. Its job is to swap other processes in and out of core memory. The swapping priority is decided on the basis of time spent in that state and event waiting for.

2.2.Exception Handling and System calls:

All the exception processing in Unix is done in kernel mode. An exception might occur because of a variety of reasons e.g. bus error, traps, interrupts from devices etc. The effect of an exception is to divert the attention of CPU from whatever it was doing and to redirect it to execute another program. Upon exception the hardware saves the program counter and status word on the kernel stack and branches to the appropriate exception handling routine,

which does the job of saving all the registers calling a general purpose 'C' routine to deal with exception. In case of exception from external devices, the appropriate device interrupt routine is called.

The exception handling facility is used in another mechanism called **system calls** by which a user process may execute a trap instruction deliberately and so obtain operating system's attention and assistance. From users' point of view the Kernel is completely defined by a description of these system calls. These system calls form the interface seen by the user. We have already talked about some of the system calls like 'fork', 'exec' , 'kill' etc. Unix provides a small but powerful set of these sytem call primitives.

2.3.FILE SYSTEM

Unix file system is a hierarchical file system. There are three different kinds of files:

1. An ordinary file is a sequence of characters. No record structure is imposed on it.

2. A directory file holds names of other files and directories and also their corresponding i-number(see below).Each directory has at least two entries: '.' and . '.' refers to the current directory itself whereas refers to the parent directory.

3. In Unix each IO device is associated with at least

one special file. The information here is not kept as a sequence of characters, instead it is provided by the device.

2.3.1."INODE" TABLE:

Every file in the file system has a unique "inode" entry identifying it. An "inode" entry contains such parameters as size of the file, its access modes, identification of the owner and his group, number of links and times of access, modification and creation. It also contains pointers to different blocks of the file. The "mode" field signifies whether the file is a regular file or special file(e.g. semaphore file).

The structure of the "inode" in the disk is similar to the in-core entry but having few more entries like time of last update etc. which are not needed for the in-core entry. The offset of a particular inode within a list of i-nodes in the disk is called i-number.

2.3.2.FILE TABLE

Every "open" file (file in active use) has an entry in the core global file table. Each entry contains a reference count, a pointer to its inode structure, and an offset for positioning the read/write head. A flag conveys the type of operation (e.g. read,write etc) that may be allowed on this file.

2.3.3.PER PROCESS OPEN FILE TABLE

The per process open file table is a part of the "u structure" . This is essentially an array of system imposed size (typical value is 2Ø), where each entry may contain a pointer to the global FILE TABLE. Many of these pointers may point to the same FILE TABLE entry.

The relationship between these tables are shown in fig.2.

2.4.INPUT/OUTPUT:

There are two different kinds of IO.

(1) Structured or **block IO**

(2) Unstructured or **character IO**

Typical example of block IO is disk and character IO is terminal. In Unix a device is identified by a device number consisting of a major number and a minor number. Major device number is same for similar devices e.g. all terminals will have same major device number. Minor device number distinguishes between two different devices of the same kind.

The block IO devices are accessed through a layer of buffering software. A number of buffers have been declared for this purpose. These buffers act like a cache for the disk storage. Each buffer is usually doubly linked into two lists;

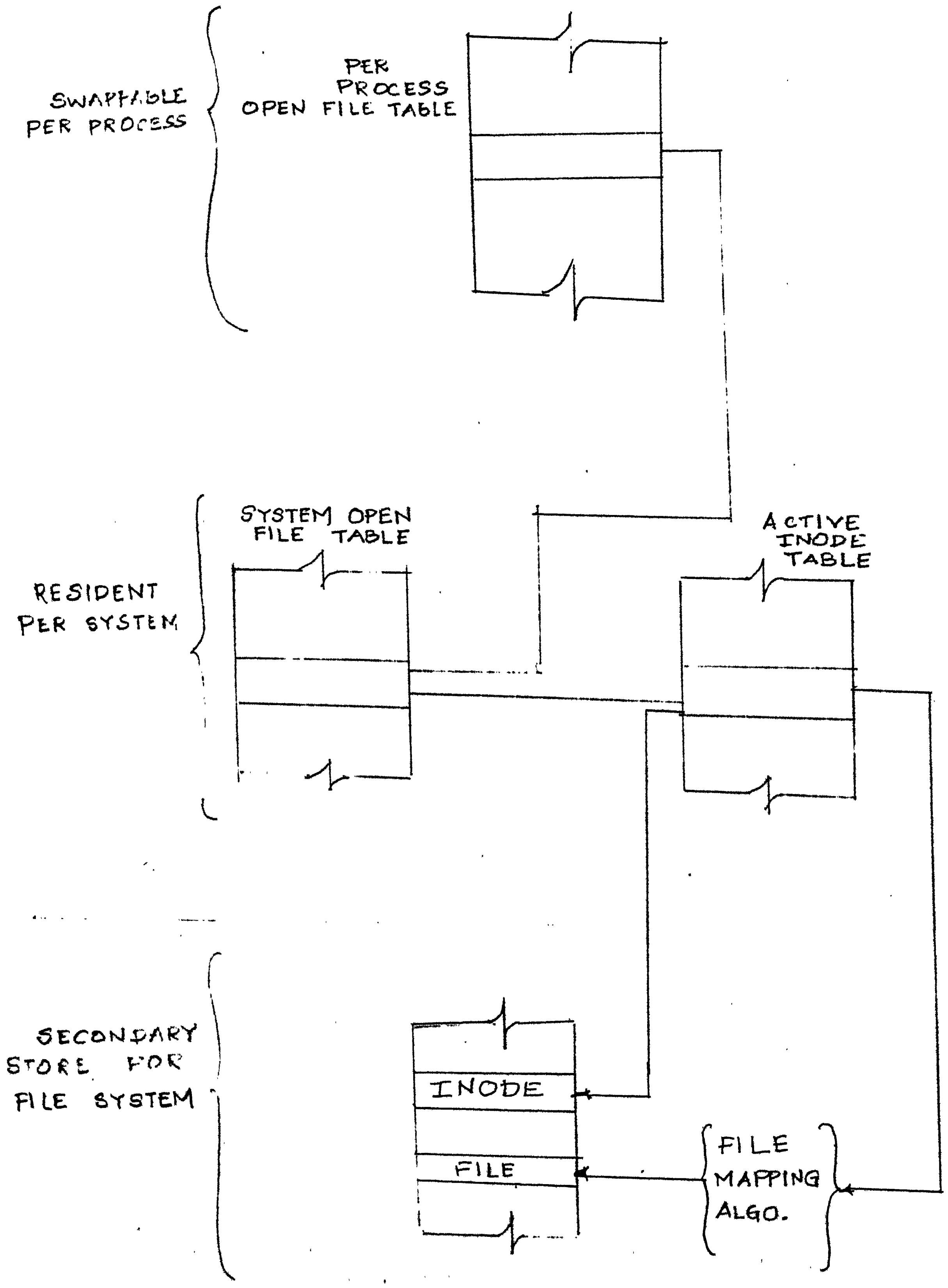

FIG. 2. FILE SYSTEM DATA STRUCTURE

1. In the device lists which it is currently associated (always)

2. In the available list for allocation for other uses. This available list is kept in the last used order.

A buffer which is on the available list is likely to be reassigned to another disk block. Normally the contents of a buffer remain unchanged till the buffer is needed for other purposes. This approach allows for delaying the physical completion of IO till absolutely necessary, thus saving lots of disk IO. But this also implies that logical completion of a block IO does not necessarily imply the physical completion also, and this might lead to some problems in the case of a sudden crash.

Character oriented peripheral devices are relatively slow, and involve character by character transmission of variable length. Associated with each terminal, which is an example of character oriented device, there is a tty structure. This tty structure along with other information keeps three character queues:

Raw queue:- Raw queue contains the characters which come directly from tty.

Canon queue:- Some processing (e.g. backspace and kill character processing) is done on raw characters and processed characters are kept in canon queue from where they are taken away by the processes waiting to read from those

tty's.

Out queue:- Out queue contains the characters which are to be displayed on terminal. The tty controller reads the characters from out queue and puts them on the terminal.

3.MEMORY MANAGEMENT UNIT OF UPTRON S-32

In this chapter we present the memory management unit of Uptron S-32, which is the important hardware aspect we are concerned with. Some of the related routines have also been briefly described.

The essential features are :

[1] The main memory is organized as segments which are paged.

[2] Each user process (referred to as context) can potentially have up to 64 segments.

[3] Each segment is divided into 16 pages.

[4] Each page is 4 Kbytes.

[5] There are 64 physical segments in the hardware.

[6] 16 contexts are possible.

Each user process is allocated a context, the context number gives the offset in the segment table. In hardware there is a 4 bit context register which points to the current context. Context zero is always used by the kernel. Moreover, if the processor is in supervisor mode then context zero is always referred to irrespective of the setting of the context register.

The logical to physical address translation process is

depicted in fig.3.. CPU outputs a 24 bit logical address . Logical address bits A23 and A22 are decoded to enable one of the following:

Register/Table	A23	A22
Unmapped	1	1
Segment table	Ø	1
Page table	1	Ø
User memory	Ø	Ø

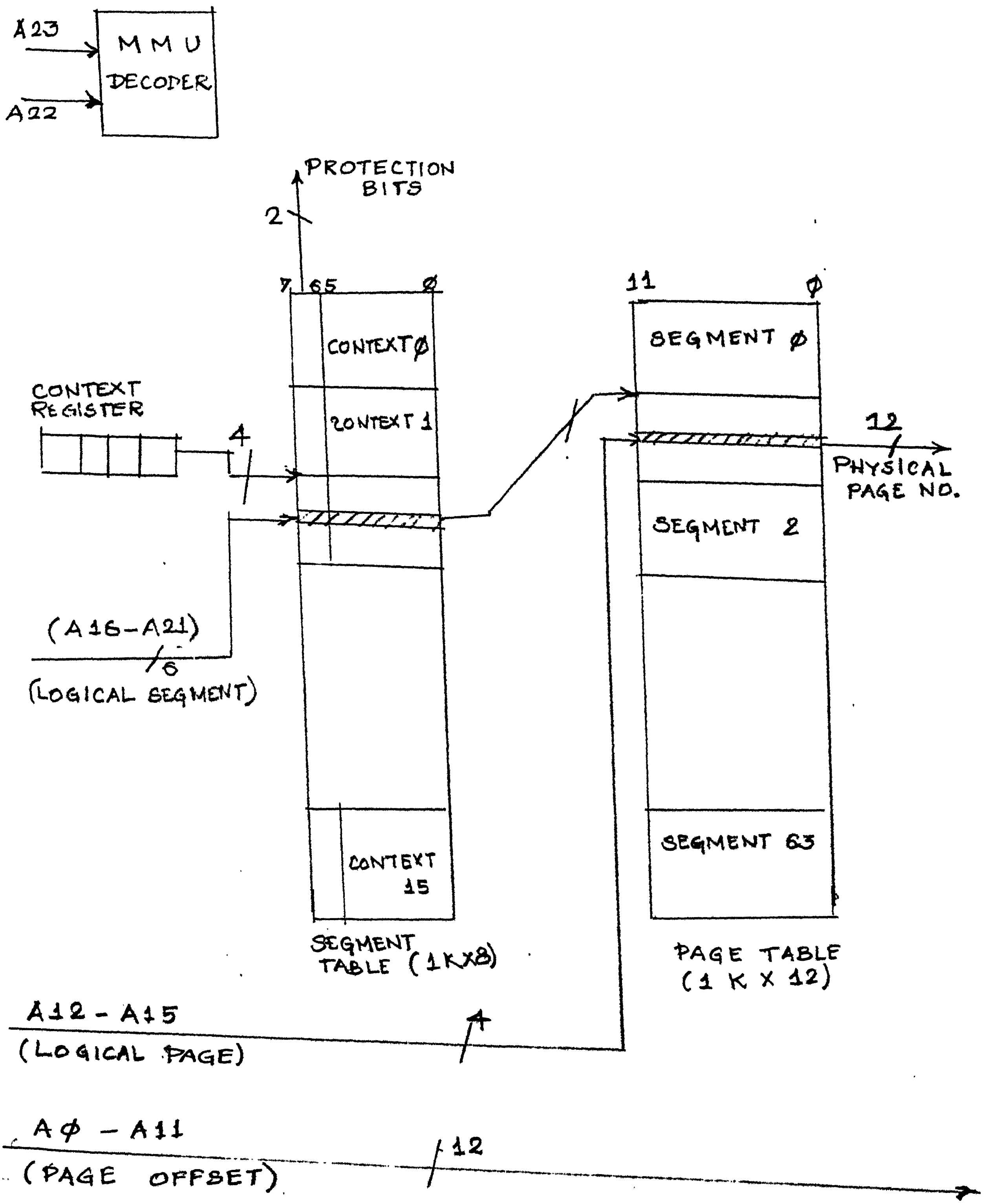

FIG. 3. ADDRESS TRANSLATION

The contents of the context register are used as an offset into the segment table to locate the segment table base address (which context). The 64 segment entries starting at this base address belong to one context. Address lines A16 through A21 form the logical segment address to address one of the 64 segment table entries.

The contents of the segment table selected by A16 through A21 form the physical segment used as an offset into the page table to locate the page table base address. Sixteen entries in the page table starting at this base address belong to the current segment. Logical address bits A12 through A15 form the logical page address in the table area located by the segment table contents.

Logical address bits AØ through A11 are used to address a specific location in a selected 4K byte physical memory page. Physical address bits A12 through A23 are outputs from the MMU which select the physical memory page. Two protection bits are also provided in the segment table.

The hardware provides only 15 contexts and less than 64 segments for the user process. Allocation and releasing of core and physical segments are taken care of by the routines 'malloc' and 'mfree'. Allocation is done using the first-fit algorithm. For managing the contexts a structure called 'context' is declared - one for each hardware context. All these 'context' structures are doubly linked up in a queue. Whenever a process requires a context a search

is made in this queue and if an unused context is found, it is removed from the list , linked back at the tail of the queue and allocated to the process requesting it. If no empty context is found then the first context in the queue is emptied i.e. resources are taken away from it. Then this context is removed from the queue allocated to the requesting process and linked back at the tail of the queue. If a process doing 'sureg' finds that all physical segments are in use then it forces the first process in the context queue to give up its physical segments.

The 'estabur' routine does just the checking of arguments passed to it while 'sureg' does the actual writing into the segment and page table for a particular context.If 'sureg' finds that the process for which it is going to set segment and page tables has already got its context set up then it simply returns. Switching from one process to other is achieved by changing the page table entry corresponding to the 'user' structure page (at address Øx3FEØØØ in kernel space).

4.DATA SHARING

4.1.Introduction:

A multiprogramming organization necessarily involves the sharing of a wide variety of resources among user and system processes. There is the sharing of hardware resources - primarily time sharing of active CPU, I/O channel, and space sharing of main and auxillary storages. Although hardware sharing was one of the original motivation for the development of multiprogramming, it was also discovered that the principal software resources could also be time- and space-shared, and that there were real benefits for doing so.

This chapter starts with explaining the need for sharing with some typical applications. The next section outlines the importance of shared memory feature in the modern technology. Following that is the brief overview of shared memory feature in some of the operating systems. In the sections to follow we shall present the user interface for our implementation of data sharing feature and the internal implementation details. Finally we shall conclude the chapter with a discussion on our implementation.

4.2.Need for sharing:

Sharing of computer programs and data is often accom-

plished by providing a user with his own private copy of the shared information. This copy could be manually incorporated into a job before submission to the computer (e.g. by insertion of a card deck) or might be automatically added by a loader referencing a library or utility file. This could still be done in a multiprogramming environment. However , several active processes often need the same code or data resident in the main storage at the same time. If each rocess had its own copy, unnecessary system costs would be incurred; these consists of the I/O overhead in loading the excess copies and the memory to store them. A serious problem occurs when multiple copies of the same data file are updated; it is almost always necessary to have one consisten file that contains all updates. For these reasons, a single copy in the main storage is frequently shared by more than one process; the sharing occurs in a time multiplexed fashion when several processes share a single CPU or essentially simultaneously when the processes are running on independent processors with common memory. We first look at some typical applications of sharing.

4.2.1.Data sharing:

a) Programs sharing a data base: Each user should be aware of the modifications done by another user . This forces us to have a single copy.

b) Several processes- possibly invoked by interactive users- simultaneously interrogating the file system: The

directories would be simutaneously in use ([SHA74]).

c) Kernel utilities: Certain utilities such as 'ps' (in Unix), which examine large kernel data structures, can be profitably implemented using memory sharing features. The standard implementation of the 'ps' uses the kernel virtual memory special file (/dev/kmem) to read the contents of the kernel 'proc' tale into an internal buffer and then examine table entries to determine the status of the processes. This approach invoves the overhead of an I/O operation, and the copying of the entire table in ps's buffers. Now, the memeory mapping features can be used to map part of ps's space to the physical memory, avoiding any copying ([SZN86]).

In general, a memory management scheme which copies page sized structures and operates in physical memory can be redesigned to simply remap these structures to avoid copying.

4.2.2.Procedure sharing:

a) System Nucleus routines: All processes share the same set of nucleus routines for example I/O programs.

b) Language processors and text editors: Popular compilers, assemblers and interpreters are often maintained in executable storage to allow their simultaneous use by more than one job. Various tools like text editors are designed in many cases to handle several terminal users at the same

time with a single copy of the system.

c) Shared libraries: Each system provides several standard libraries consisting of functions which can be linked to use programs prior to run-time. It would be advantageous to have all running programs access a single in-core copy of this library code, rather than using up memory and disk resoources to give each program its own private copy.

A detailed study on procedure and data sharing can be found in [SHA74].

4.3. Importance of shared memory feature:

This section briefly outlines the importance of shared memory feature in modern technology, mainly in multiprocesor systems.

Shared memory has long been regarded as the most flexible and powerful way of sharing data among parallel processes ([AHU86]).

One of the main type of architectural approaches to the design of computer for AI applications is : Multiprocessors that support interactive MIMD operations through shared memory space([HWA87]).

Many physical problems and abstract models are seriously compute-bound. Since sequential computer technology faces seemingly insurmountable physical limitations, it is widely believed that the only flexible path toward higher

performace is to consider radically different computer organisations in particular exploiting parallelism. One of the most promising and the simplest type of parallel machines is the well known multiprocessor architecture, a collection of autonomous processors with either *shared memory* or *distributed memory* that are interconnected by a homogeneous communication network ([HAM86]).

Lundstorm et al. ([LUN87]) point out the extensive use of very large shared memory, for the design of a very large, very high speed multiprocessor systems. They have presented a Flow Model Processor (FMP) conceptual design , which contains many free-standing processors, each of which is capable of independent execution sequences. Each processor contains a local memory, local cache etc. Each processor has access to *shared memory* resources, which consisted of as many memory modules as there were processors. The processor accesses the shared memory through an interconnection network. The processors share a common code memory from where the program code is transferred to the local program caches. In addition to this processes share a "data base memory" through interconnection network. The system was intended to solve large scientific problems.

Parallel processors are mainly divided into two groups - farms and cubes. A typical farm comprises a small number of processors, generally no more than eight, each usually like a minicomputer in performance. The processors communi-

cate through *shared memory* accessed over a bus or a network of direct processor-to-processor connections. On the other hand, cube will normally include large number of processors, each having their own local memory and communicating over a network ([MOR87]).

Linda, a parallel programming language, supplies a form of logically-shared memory without assuming any physically shared memory in the underlying hardware ([AHU86],[CAR86]). Standard communication protocols require that information be handed from process to process; no process can unburden itself of new data without first determining where the data should go, and then handing it along explicitly. Linda's proceses, on the other hand, are anonymous drones sharing access to one data pool. Shared memory has long been regarded as the most flexible and powerful way of sharing data among parallel processes - but a naive shared memory is hard to implement without hardware support, and requires the addition of sysnchronization calls if it is to be safely accessed in parallel. In Linda, however, the shared memory's cell-size is the logical tuple, not the physical byte and so, it is coarse-grained enough to be supported efficiently without special harware. And because, in Linda's shared memory, data may not be altered in situ - it being accesible via read, remove and add instead of the standard read and write - it may safely be shared by any number of parallel processes([CAR86]).

4.4. Memory sharing feature in various operating systems:

The following section gives an overview of memory sharing feature available in various operating systems.

4.4.1. Unix System V:

In sysuem V, the shared memory feature allows processes to share parts of their virtual space.

The "shmget" system call creats a new region of shared memory or returns an existing one, the "shmat" system call logically attaches a region to the virtual address space of a process, the "shmdt" system call detaches a region from the virtual address space of a process, the "shmctl" system call manipulates various parameters associated with shared memory (i.e. ownership, mode etc.). Processes read and write the shared memory using the same instructions they use to read/write regular memory. After attaching shared memory , it becomes part of the virtual adddress space of the process, accessible in the same way as other virtual addresses; no separate system calls are needed to access data in the shared memory.

Some of the advantages/problems in the sysv implementation are :

[1] Highly flexible and easy to use.

[2] Processes can share different portions of the same region with different access rights.

[3] A questionable naming scheme: To creat a shared memory region a numeric key is used . This may lead to unwanted situations.

[4] The identifier returned to the user for further use is an index from the global table. Thus it is not providing a consistent and uniform interface. Returning a global table index may lead to inconsistency of data structures by a mischievious programmer or even by erroneous programming.

[5] Since the creation of the segments are done only at the run time there is no way to have initialized segments.

For system call descriptions reader may refer to [SYSV]. Implementation details can be found in [BAC86].

4.4.2. 4.2 BSD

4.2 BSD system supports sharing of data between processes by allowing pages to be mapped into memory. These mapped pages may be shared with other processes or private to the process.

```
mmap(addr,len,prot,share,fd,pos)
```

causes pages starting at 'addr' and continuing for 'len' bytes to be mapped from the object represented by 'fd' at absolute position 'pos'. The parametter 'share' specifies whether modifications made to this mapped copy of the pages are to be kept 'private' or to be shared with other references. The parameter 'prot' specifies the accessibility of

the pages. The addr, len and pos must be multiples of page size.

A process can move pages within its own memory by using "mremap" call.

mremap(addr,len,prot,share,fromaddr)

This call maps the pages starting at 'fromaddr' to the address specified by 'addr'.

A mapping can be removed by the call

munmap(addr,len)

This causes further references to these pages to refer to private pages initialized to zero.

'fd' must represent a character special file capable of mapping pages.

Interested reader may refer to [UNIXa] for details.

4.4.3. UNOS:

Unos, provides two system calls for sharing segments.

"segshare" system call attaches to a shared segment. The format of the call is

segshare(uaddr,fdown(file),offset,icount,size,mode)

The segment starts at the 'offset' in that file and is 'size' bytes long, 'uaddr' is the user preferred starting address in the process's virtual address space. 'icount' (initialization count) indicates how the segmenat should be initialized. 'mode' specifies access mode.

Initialization can be done in one of the following ways:

[1] Simply clear the segment to all zeroes by using an 'icount' of Ø.

[2] Read in from the file a specified number of bytes which will form the first part of the segment. If there is not enoungh data in the file to fill the segment, the remaining is filled with zeroes. If there is more in the file than can fit in the segment, the routine that initializes the segment stops reading the file.

Shared text objects are implemented as shared data segments which are globally read only, having an offset equal to the size of the executable header, and length equal to the size of the text of the executable image. A process may attach to the shared text using this.

"segunshare" releases a shared data segment.

Reader may find the details in [UNOS].

4.4.4. EMBOS:

EMBOS is a message based operating system running under ELXSI machines([ELX84]).

EMBOS provides a process oriented environemnt in whixh each process, including all system processes, executes in its own distinct virtual address space. All communication between processes is transmitted via hardware supported inter process messages. Alternatively blocks of data may be

transferred between processes as *shared pages*, with synchronization provided by interprocess messages.

"Memory Manager" process maintains within its witrual space the virtual-to-physical maps for each EMBOS process including itself. The required adress translation for an executing process are then brought into CPU registers by dedicated microcode which has access to the page maps in memory.

One of the important functions supported by the Memory Manager is that of two or more processes sharing physical pages, whereby a shared physical page may occupy different virtual addresses in the sharing processes. Sharing of pages between processes is established by the memory manager when a process sends a group of pages to another process, while retaining access to the pages. Access to the shared pages is synchronized by the processes via interproces messages.

4.4.5. MACH:

A highly sophisticated virtual memory management scheme is available in Mach, a multiprocessor operating system kernel currently under development at CMU ([ACC86]).

Virtual memory design allows tasks (an abstract concept like processes) to

(a) allocate regions of virtual memory

(b) delallocate regions of virtual memory

(c) Set the protections on regions of Virtual memory

(d) Specify the inheritance of regions of Virtual Memory.

Shaed memory is created in a controlled fashion via an inheritance mechanism.

In Mach, when a "fork" is invoked, a new(child) address map is created based on the old(parent) address map's inheritance values. Inheritance may be specified as 'shared' ,'copy' or 'none' and may be specified on a per-page basis. Pages specified as 'shared' are shared for read and' write access by both the parent and child maps. Otherwise copied if 'copy' or not at all inherited if 'none' is specified. If 'none' the child's corresponding address is left unallocated.

4.4.6. UTek:

UTek is Tektronics enhanced version of 4.2 BSD running on Tektronix 6000 series machines. The virtual memory implementation is substantially different from that of 4.2 BSD, though the system calls look very similar in nature ([SZN86]).

Following is the brief description of the system calls:

[1] mmap(pid,fromaddr,toaddr,len,prot,share)

The mapping routine 'mmap' allows a process to access areas of other processes through its own address space. It

causes calling process's pages starting at 'toaddr' and continuing for 'len' bytes to map onto the process with id 'pid', starting at the objects's pages 'fromaddr'. If the parameter 'share' is true, both mappings will share the same memory. Otherwise, a private copy of these area is made, and changes through one mapping are not visible to the other.

[2] mremap(fromaddr,toaddr,len,prot)

"mremap" is used to move a mapped area within the address space, possibly with new protections. This may include multiple mappings, and unmapped areas.

[3] munmap(addr,len)

"munmap" causes the process pages starting at 'addr' and continuing for 'len' bytes to be removed from the legal address space of the process.

4.4.7. XENIX - 3

Xenix-3 supports shared memory between any number of processes. The shared memory is called a 'segment'. There may be several shared segments, each shared between a subset-of the active processes. A process may access several shared segments.

In Xenix-3, a process must map and unmap the segment each time it needs to use it. This is because it is forbidden to execute any system calls while the segment is mapped in. Any other system, we have described, do not impose this constraint and so system calls can be executed

even if the shared segments are mapped.

Xenix-3 provides a simpler set of system calls to deal with shared memory. Here a zero-length file is used to represent shared memory.

There are two system calls "sdget" and "sdfree" which allows processes to create and attach to a shared segment, free an attached segment respectively. There are two special system calls "sdenter" and "sdleave" which allow processes to access the mapped area and to leave the mapped area after access operations are over. By "sdenter" system call one can make sure that no one else is accessing the area at the same time. There are two another peculiar system calls "sdgetv" and "sdwaitv" which gets the version number , which is incremented each time access is made to this shared memory, and waits for a version number respectively.

For more details reader may refer to [ROC].

4.4.8. DOMAIN:

Domain operating system, running in Apollo computers, provides sharing of data through mapping ([APO85]).

In Domain, mapping associates part of the user address space with a disk file .When the user maps a file, the system reserves part of the address space for the file. In addition, system returns a pointer with which user can access the file . As the user references different parts of

the file, the system brings ,the required pages to the memory.

Files that are mapped should contain data in user-defined format. So, when a user maps a file, he has to make sure that the program understands file's format.

If many programs on the same node map the same file, they share the same memory for the same pages of the file. Therfore, these programs can simultaneously map a file for reading and writing. If one program writes to the file, others can access the changes. However, user must provide his own sysnchronization mechanisms when several programs write to the same file.

The important feature of the map system calls of the Domain is 'locking facility'. Two parameters namely 'concurrency' and 'access' determine the type of lock that user is requesting. The concurrency indicates the number of programs that can access a file, while 'access' indicates the types of access that each program can have. Depending upon the lock provided by the first user other users may be able to map to the same file. If the locks do not agree, the system call will return with an error.

A detailed description regarding these system calls can be found in[APO85].

4.4.9. MULTICS

Multics ([DAL68]) was one of the early operating systems to employ "dynamic sharing" based on "dynamic linking" strategy, where resolving the references is postponed till the last possible moment (i.e) only when it is referred during the execution for the first time.

The MULTICS solution to the problem of dynamic sharing involves the use of two types of interfaces.

[1] A "Linkage section" is defined for each potentially sharable or sharing segment. Transfer of control and external data references occur by indirectly addressing through linkage sections. Each process receives a private copy of the linkage section when it uses a segment.

[2] A private stack segment is allocated to each process for storing arguments, return addresses, processor states and temporaries.

All addresses are kept symbolic in the procedure so that many processes can share it simultaneously. 'Linkage section' is the place where the run-time addresses are kept. MULTICS uses a 'trap enable' technique to effect this symbolic address to run-time address translation.

A detailed study on MULTICS can be found in any one of the following ([SHA74],[DAL68],[SAL78]).

4.5. Design Issues

In this section, we shall discuss some of the decisions taken while implemnting shared data segments.

Before discussing the decisions taken it is worthwhile to have a look at the hardware limitations which led us to take some of the decisions given below.

In Uptron S-32, the protection is provided at the segment level (64K). Immediate observation of this fact is that we can not have more than one data segment per logical segment of the virtual space if the segments' access modes are not compatiable. But the present implementation allows only one shared segment per logical segment of the virtual space.

Since the protection is at the segment level there is going to be lot of virtual address space wastage. If the hardware provides protection at the page level this wastage could be considerably reduced.

The minimum allotted size of the segment is always a page size (4 K in S-32). This allows the user to access some area which he did not request at all. Of course, in any hardware, where memory is allocated in pages it is not possible to avoid this wastage.

Our main idea was to provide two types of shared segments.

[a] **Unbounded segments:** These segments can be attached any where in the virtual space. So these segments do not

contain any references (i.e. pointers).

[b] **Bounded segments:** These segments have got to be attached only to a particular virtual address. This is necessitated by the fact that we may have situations wherein the data segments may contain pointers pointing to other elements in the segment. A typical example could be sharing a linked list of elements.

We wanted to provide a facility to the user whereby he can initialize the data even before attaching to it. This is useful if we want to attach to a segment which has always some initial value. This led us to create segments outside the kernel.

Once we decided to have the segments as static entities, our obvious choice was to handle them as files. This makes many things easier for us like deleting the segment, changing the permission mode etc. Now let us have a look at the advantages and disadvantages of this method.

One of the main advantages of having shared segments as static entities is a better naming scheme, compared to other systems which allocate the segments at run time (e.g. In Unix System V , a numeric value is used as a key).

Creating the segments outside the kernel allows the user to initialize the data segment prior to run time. This is an advantage over an implementation in which the creation of segments are done only at the run time.

If the creation of the segments are done only at the run time, the virtual address with which one user attaches to a segment is invisible to the other user, which leads to fact that it is not posssible to share a 'bounded segment'.

The main drawback of this scheme is that it is more restrictive in nature. We are compromising flexibility to gain advantages mentioned earlier. What can be achieved easily by a scheme in which the creation of segments are done at the run time , we are doing it with some constraints. This is acceptable since we provide a simple interface to the user.

4.6. User interface

In this section we shall present the shared segments from the users' point of view.

4.6.1. Creation of the segment

The following restrictions are imposed:

[1] User has to describe a shared segment under a single structure. This forces the user to access a shared segment through a single pointer which makes things simpler. Let this structure be available in some file (say "st1.h"). User should creat a .c file (say "test1.c") in the format given (see APPENDIX I). The initialization ,if any, has to be done only at this file.

--

[3] Now the user has to call the routine "/etc/mkshare" to create a shared segment. A typical call could be

/etc/mkshare shar1 test1.c vaddr

where "shar1" is the user wanted name of a shared segment, "test1.c" is the file created according to step 2 and "vaddr" is the virtual address . This virtual address is needed only if the segments are bounded (i.e. the segment contains some pointers). A typical situation where this virtual address could be used is when the user wishes to have a linked list of elements.

4.6.2. System calls

We have provided two system calls one for attaching to a shared segment and another for detaching a segment.

[1] **SHMATT:** The format of the system call is

shmatt(path,vaddr,perm)

attaches the segment denoted by 'path' to the virtual address 'vaddr' with the access right 'perm'. If 'vaddr' is Ø then the kernel looks up for a suitable area . If the call is successful this call returns the actual virtual address where this segment was attached.

[2] **SHMDET:** This system call detaches the shared segment from the process' virtual space.The format of the call is

shmdet(vaddr)

where 'vaddr' is the "shmatt" returned virtual address.

Detailed description of these system calls can be found in Appendix II.

```
    /*
     * This programs just illustrates the use
     * of shared segment system calls.
     */

#include <data.h>
main()
{
  int p,i,*k,*q;
  extern shmatt(),shmdet();
  char *c;

    /*
     * Attach to the shared segment "shar".
     * Assume "shar" has already been created.
     * Let this be an array of integers.
     * (see Appendix I for creation of shared
     * segments)
     */

       if((p = shmatt("shar",Ø×Ø,DWRITE)) == -1)
         printf("unsuccessful shmatt");
       else {
           printf("return add %o",p);
           q=(int *)p;
           k =q;

    /*
     * Write into the segment
     */
        q = k;
        printf("write starts");
        for(i=1ØØ;i<2ØØ;i++) {
             *q++ = i;
        }
        printf("final add write %o  ",q);

    /*
     * Read whatever has been written now
     */
        printf("start read add %o  ",k);
        for(i=Ø;i<1ØØ;i++)
        printf("%d ", *k++);
        printf("final add read %o",k);
    /*
     * Detach the segment
     */
       if((i = shmdet(p)) == -1)
            printf("can't detach the segment");
       else
            printf("successfully completed");
    }
}
```

4.7. Implementation details

In this section we shall talk about the details regarding how shared segment feature has been implemented.

One of the main ideas we kept in mind while deciding the implementation details was to organize the shared segments in a very similar manner to that of text segments. So, we decided to have a global table to handle shared segments. We shall now look at the data structures followed by the algorithms for 'shmatt' and 'shmdet'system calls.

4.7.1. Data structures

[1] DATA TABLE: This global system table essentially contains an entry for each shared segment currently under use. An entry in this table contains the size of the segment,counts to indicate total number of references and total number of loaded references, core and disk addresses and a pointer to the associated inode. Whenever a new segment is to be attached by a process an entry from this table is allocated.

[2] Every process should contain information regarding the shared segments it has got attached to. Since this information will be used by the swapper, this information has to be there in the core irrespective of whether the process is in the core or in the swap space. So, we have to essentially keep this information in the 'proc' structure (see chapter.2). An array of

structures were made part of the 'proc' structure. The length of the array gives the maximum possible number of segments that can be ttached to this process. which is tunable. In the sequel we shall call this array as 'p_data' array. Each element of this array contains the actual virtual address where this process has attached to a particular segment, pointer to the 'DATA TABLE' entry corresponding to this segment and the type of access (e.g. READ ONLY etc.).

4.7.2. Algorithms

In the next few pages we have outlined the algorithms for 'shmatt' and 'shmdet' system calls. The names of the routines which accomplish a step of the algorithm have been given within parenthesis in capital letters.

Algorithm shmatt:

input: segment name, virtual address, access mode

output: actual virtual address where the segment is attached

```
{
     Get a free slot from the 'p_data' array ;

     Convert the segment name into inode pointer ('NAMEI');

     Make validity checks by reading  the  header  from  the
     inode pointer ('READI');

     If the segment is bounded take the bounded address from
     the  header  irrespective of the user specified virtual
     addr and if it does not lie in the page boundary return
     with error;

     Otherwise make the use specified virtual address to lie
     on a page boundary;

     If the virtual address is zero look up for  a  suitable
     space ('ALLOT');

     Otherwise check whether the virtual  address  range  is
     free ('CHECKVADDR');

     If the virtual address is legal, call algorithm  'DAL-
     LOC';

     If no error return(actual virtual address);

      }
```

Algorithm : dalloc

input: inode pointer,size of the segment,'p_data' element

output: none

{

If there exists an entry in the global 'DATA TABLE' and core count is not zero hook up to it, return after setting the segment table , page table entries ('SUREG'); Initialize it in the core if it has not been initialized so far.

There exists an entry in the 'DATA TABLE' and the core count is zero then check whether enough core is available. If enough core is available and there is data in the swap area copy into the core. Otherwise copy from the file. If enough core is not available swap it out. When it comes back enough core will be available and do the initialization and setting etc, return.

Get a new entry from the 'DATA TABLE' and initialize the entries;

check whether enough core is available. If enough core is available initialize the core from the file Set the mappings return; If enough core is not available swap it out. When it comes back enough core will be available and do the initialization and setting etc.

Algorithm shmdet:

input: virtual address

output: Ø on success and -1 on error

{

Check the 'p_data' array to check whether the given virtual address is a 'shmatt' returned virtual address.

If not, return with error;

Decrement the reference count of the 'DATA TABLE'entry If the number of loaded references drops to zero copy it into the swap area, if this virtual address was not due to 'shmatt' call with READ ONLY access and swap area has already been initialized. Free the core and swap areas if necessary ('DFREE','DCCDEC').

Undo the mappings, return;

}

4.8. Implications

In this section we shall look at the impacts of this implementation on the rest of the code.

One of the main areas where much attention was given is the memory management unit. Understanding the memory management unit itself was the first task. We have described the memory management unit (mmu) in chapter 3. Once we understood the mmu our job became much simpler.We have modified the mmu to handle the shared data segments. Allocating page table entries, setting the mappings in segment and page table entries are the important functions served by this unit.

The next problem was to decide regarding the problems associated with swapping and the initialization of shared data segements both in the core and swap space. For this purpose we have introduced two flags, DSNOCORE and DSNOSWAP , which essentially tells that the core space and swap space ,respectively, has not been allocated for a particular shared segment. We shall describe the situations where these flags are needed.

[a] When the swapper chooses a process for swapping in it makes sure that enugh space is available for this process to execute.It should also make sure that space is not allocated twice for the same shared segment.This can simply be achieved using the reference counts in the 'data table' entry. Suppose, a process, say P1 ,

is trying to attach to a shared segment and let us also assume that this is the first time this shared segment is being attached by any process. If enough core is not available the process swaps itself out and the swapper ('SCHED') will make sure that enough space is available when P1 comes in again. If the swapper chooses P1 , it would have allocated enough space for P1. P1 initializes the core count (which gives the number of processes referring to this segment in the core) and waits for the CPU. In the meantime if some other process, say P2, tries to attach the same segment it will find that some other process is already referring to this segment in the core and hence will try to access the contents of the segment, which obviously should be avoided.To prevent this type of invalid accessing of core contents , we use the DSNOCORE flag.

[2] The second flag DSNOSWAP comes into picture in a very similar situation. But the difference here is that P1 might have incremented the total reference count (which indicates total number of processes referring to this segment both in the core and the swap space) as soon as it tries to attach to a segment but might have got itself swapped out. So, P2 might try to take the data from the swap space, which has not been initialized so far. To avoid this we use the above mentioned flag.

We wanted the initializing of data in the swap space to be done only when need arises. Copying the core contents

into swap space every time whenever a process detaches the segment or it is being swapped out is unnecessary. We achieve this in the following way. When a process wants to dereference a shared segment either by swapping or by detaching, we check whether this is the last process referencing to it in the core and there exists some process which is referring to this still in the swap space. If so, we copy the present core contents of the segment into swap area. We have achieved one more level of saving by not copying the contents if the dereference made corresponds to a 'shmatt' system call with READ ONLY access mode and the swap space has already been initialized.

One of the system calls which will be affected due to the implementation of the shared segments is 'brk' system call. This system call expands or contracts the data region of the process image to a user requested value. Since shared segments become part of the address space after attaching it is necessary to check that the new value requested by the user in the 'brk' call does not collide with any of the virtual addresses of the shared data segments. But this has got one major drawback. Assume a process attaches to a shared segment immediately after the data region (assume we have taken care of the boundary problems). Also assume that this segment is held attached throughout the life of the process. This implies that whenever the process makes a request for expanding the data region it may return with an error, because the virtual addresses may col-

lide as stated above. Our suggestion is to allocate a suitable space if one available and return the virtual address to the user. But this will mislead the user if he thinks that the area he has obtained is contiguous with the data region in the virtual space.

The routine 'grow' (which essentially expands the stack space) has to be handled in a similar manner described above. However, here it is a must that the stack should be contiguous in the virtual space.

We have taken care of the shared segments in other system calls and routines in a manner very similar to that of text segments. Some of the routines which have been modified in this manner are: "exit" (terminate a process),"swtch" (CPU scheduler) etc.

In the swapping routine ("SCHED") we have handled the shared segments similar to text segments except for a minor modification which is due to the fact that we have to check whether there exists a copy of this segment in the swap space or not, when the segment is not in the core. In the case of text segments, whenever a new text segment comes in for the first time it is initialized in the swap space. Here this is advantageous because the text is a READ ONLY code and so we need not swap it out every time whenever a process is swapped out.

"dosureg" (which sets segment and page table entries

for a user process) is the one which has to be handled very carefully and which in some sense is the core of our algorithm.

4.9. Conclusion

We shall conclude this chapter by presenting a comparision of our implementation with the implementations of other operating systems and also pointing out some of the possible future expansions.

[1] our implementation provides a way for many processes to attach to a segment and a process to many segments, like other systems described.

[2] We expect the users to provide their own synchronization tools like most other systems (e.g. Unix System V). For this purpose we have implemented semaphores (see chapter 5).

[3] Unlike other systems described, we initialize the data outside the kernel. As we had already described, the main use of this is to provide a way for the user to initialize the data prior to run time and to have bounded segments.We provide a much simpler interface through this. The main drawback of this is that it is inflexible. User do not have the control over the size of the segment. Users cannot share a portion of the segment.

[4] Each user is free to map to any segment. One user can

in no way stop other user from mapping to a segment. This is possible in DOMAIN operating system, where if a user gets a segment with EXCLUSIVE WRITE access, other users cannot map to that segment. Of course, we should keep in mind that our implementation checks for file permissions before attaching a segment to the process.

[4] The data segment can contain any type of data structure. In DOMAIN operating system, the data segment is assumed to contain data in a user defined format throughout the segment.

Following are the possible expansions that can be done based on our implementation.

[a] In the present implementation we have implemented only one shared segment per logical segment of the virtual address space. This can possibly be extended to include more than one segment if their protections are same.

[b] We can provide an option by which the segments can be created at run time.

This implementation is straight forward. Once we do that we might be able to exploit the advantages of this method,which we mentioned earlier..

In conclusion, this implementation exercise gave a good insight into the memory management unit , process management routines and other associated routines.

5. SEMAPHORES

5.1. Introduction

This chapter starts with presenting the notion of concurrent processes along with other definitions required for our discussion. Then some important synchronization tools like semaphores, monitors etc. are presented with their merits and demerits, which explains why we chose semaphores over other tools. Following that is the brief description of IITKIX semaphore implementation from the user's point of view . Finally, the internal details of impelmentation are presented followed by the conclusion.

5.2. Definitions

5.2.1. Concurrent Processes:

A process, as we have already defined, is simply a program in execution. Processes are *concurrent* if their executions overlap in time. Whether individual operations of concurrent processes are overlapped or interleaved in time, or both, is irrrelevant. Whenever the first operation of one process is started before the last operation of another process is completed, the two processes are concurrent ([BRI73]).

The concepts of synchronization and mutual exclusion

attains significance in the light of interacting processes - concurrent processes which have access to common variables.

Common variables are used to represent the state of physical resources shaed by *competing processes.* They are also used to communicate data between *cooperating processes.* So, in general, all common variables represent shared objects called resources. Interacting processes therefore can also be defined as processes which share resources.

5.2.2. Mutual Exclusion

System resources may be classified as sharable , meaning that they may be used by several processes concurrently or non-sharable meaning that their use is restricted to one process at a time. The mutual exclusion problem is that of ensuring that non-sharable resources are accessed by only one process at a time.

5.2.3. Synchronization

Generally speaking the speed of one process relative to another is unpredictable, since it depends on the frequency of the interruption of each process and how often and for how long each process is granted a processor. We say that processes run asynchronously with respect to each other. However to achieve successful cooperation there are certain points at which processes must *synchronize* their activities. Synchronization is a general term for any constraint

on the ordering of operations in time.

5.2.4. Critical section

"Critical section" is a segment of code, in which the process may be reading common variables, updating a global table, writing a file and so on. The important feature of the systme is that when one process is executing in its critical section , no other process may be allowed to enter into its critical section. Thus the execution of critical sections by the processes is mutually exclusive in time ([PET]).

5.2.5. Deadlock

A set of processes is in a *deadlock* state when every process in the set is waiting for an event that can only be caused by another process in the set ([PET]).

5.2.6. Starvation

"Starvation" is a situation where a process is indefinitely blocked. A process which is ready to run but lacking the cpu can be considered blocked, waiting for the cpu. An unfair scheduling strategy may lead a process to starve ([PET]).

5.3. Implementation tools:

Some of the mechanisms to achieve mutual exclusion and synchronization are briefly discussed in the following section. Their merits and demerits are also pointed out.

5.3.1. Semaphores

The oldest, and possibly simplest, mechanism -semaphores- was first suggested by Dijkstra. He introduced two primitive operations that considerably simplified the communication and synchronization of processes. In their abstract form , these primitives, designated P and V, operate on the non-negative integer variables called *semaphores.* Let S be such a semaphore variable. The operations are defined as follows:

[1] **V(S):** S is increased by 1 in a single *indivisible* operation; the fetch, increment and store operations cannot be interupted, and S can not be accessed by another process during the operation.

[2] **P(S):** Decrement S by 1, if possible. If S=Ø then it is not possible to decrement S and remain in the domain of non-negative integers; the process invoking P operation waits until it is possible. The successful testing and decrementing of S is also an indivisible operation ([SHA74]).

When a semaphore can take only the values Ø or 1, it is termed as a *binary* semaphore. If it takes any non-negative value then it may be termed as *general* semaphore.

The main advantages of semaphores are

[1] Implementing semaphores is relatively simpler than implementing other sychronizing tools.

[2] The semaphore solution is sufficiently general for all synchronization problems([LAG78]).

[3] It is sufficiently flexible to express arbitrary scheduling disciplines([LAG78]).

The main drawback with semaphores is as follows. Semaphore operations must be used with great care. Programs that use semaphores are notoriously difficult to verify; programming errors may lead into deadlocks or inconsistent situations which are difficult to analyze.

However, semaphores have been successfully used as basic synchronization tool for complete operating systems ([DIJ68]).

5.3.2. Conditional Critical regions:

A synchrnization tool that is, in a certain sense, at the opposite end of the spectrum from Dijkstra's solution, was introduced by Brinch Hansen ([BRI73]) in the form of "conditional critical regions". There, the aspects of mutual exclusion and of intermediate blocking are taken care of by purely syntactic means:

-The construct

"region v do .... end"

associates to the data object v an (anonymous) mutex semaphore, which may lock all "region v" statements.

-Every access operation of the data object is a "region

v" statement.

- The clause "Await B" inside "region v" statement temporarily suspends the execution of the calling process if and so long as the condition B on the state variables is false; while the process is blocked, the lock associated to v is open. When the execution of the process is resumed, B is true and the lock is set.

This synchronization tool allows programmer to express the synchronization conditions directly; But the drawbacks are:

[1] Implementation is quite expensive. The problem is that the conditions B have to be evaluated very frequently.

-for avoidance of deadlocks: at least if a process leaves a region or is blocked by an "await" clause;

-for minimal blocking: after every change of state variable.

So the overhead is very heavy.

[2] Since it becomes a part of the language, the compiler has to be modified.

[3] Fair scheduling strategies cannot be obtained ([LAG 78]).

[4] Nesting of conditional critical regions may lead to unwanted sequentializations.

5.3.3 Monitors:

In order to avoid the inefficiency associated with coniditional critical regions , Hoare and Brinch Hansen independently introduced the concept of *monitors.* Here the access operations of a data object are collected in the syntactic context of a type definition and are again locked collectively by a common anonymous lock variable. The mutual interaction, however has been made explicit by the operation *wait* and *signal*

-The operation 'wait(q)' blocks the calling process and puts its process identification on a waiting queue q; the lock on the data object is released.

-The operation 'signal(q)' chooses the process from the queue q and activates it; as the lock remains set, the calling process is suspended until the newly activateed process leaves the monitor or is blocked again.

The monitor concept, if used with care, can lead to very efficient implementatios, however, it suffers from the same deficiency as conditional critical regions; if monitors are nested, process may be blocked unnecessarily. Moreover the verification of programs using monitors may be difficult. The situation becomes worse if explicit shceduling strategies are incorporated. In addition implementation is also difficult.

5.3.4. Eventcounts:

This is not based on the concept of mutual exclusion, rather on observing the sequencing of significant events in

the course of an asynchronous computation. An *event count* is a communication path for signalling and observing the progress of concurrent computation. Event counts are more natural mechanism for controlling the sequence of events that do not need mutual exclusion ([REE77]).

An *event count* is an object that keeps a count of the number of events in a particular class that have occured so far in the execution of the system. An event count can be thought of as a non-decreasing integer variable. Some of the primitives provided to operate on these event counts are : a) *advance* which signals the occurence of an event associated with a particular event and b) two primitives, *await* and *read*, that obtain the "value" of an event count.

Detailed descriptions about these tools can be found in any of the following ([BRI73],[PET],[SHA74]). A comparative study of different synchronization tools may be found in ([LAG78]).

Hence mainly due to the simplicity in implementation we have chosen semaphore over other mechanisms.

5.4. Synchronization tools in various systems

This section briefly outlines the various synchronization facilities available in some of the operting systems.

5.4.1. UNIX SYSTEM V:

System V uses semaphores for synchronization. In Sys V the semaphore calls allow process to synchronize execution by doing a set of operations atomically on a set of semaphores. These semaphore system calls are a generalization of Dijkstra's P and V operations, in that several operations can be done simultaneously and the increment and decrement values can be greater than 1. The kernel does all operations atomically; no other processes adjust the semaphore values until all operations are done. If the kernel cannot do all the operations, it does not do any; the process sleeps until it can do all the operations. The need for this generality arises due to the fact that when a process tries to acquire multiple resources deadlock can occur.

The *semget* system call creats an array of semaphores. The kernel allocates an entry in the global semaphore table that points to an array of semaphore structures. The index of this entry is returned to the user as the semaphore identifier. Processes manipulate semaphore with *semop* system call. This call allows a wide variety of operations to be done on a set of semaphores. The *semctl* system call contains a myriad of control operations for semaphores, such as removing a semaphore table entry.

The major advantages of this scheme are : [a] A separate name space and so no per process resource is used. [b] Ease of use

Interested reader may refer to [SYSV] for a detailed

description on the system calls. The implementation details can be found in [BAC86].

5.4.2. UNOS OPERATING SYSTEM:

In UNOS, the operting system in our SUNRAY machine, event counts are the synchronizing mechanism employed.

In UNOS the event count consists of a pair of integers. Each portion can be separately read and advanced. So, using this effectively we can creat a sort of lock , which can be used for mutual exclusion and sysnchronization.

Details regarding the event count system calls can be found in [UNOS].

5.4.3. XENIX-3:

Binary semaphores are employed as the sysnchronization mechanism in XENIX-3. In Xenix-3 a semaphore is a new kind of file with an inode, but no data. Only read permission is used. If a process has it, it can open, acquire and release. The interested reader can refer to [ROC] for a brief description on the system calls.

5.4.4. "THE" MULTIPROGRAMMING SYSTEM:

The whole system essentially consists of sequential processes. Their harmonious cooperation is regulated by means of explicit mutual sunchronization statements. Explicit mutual synchronization of sequential processes is imple-

mented via "semaphores"([DIJ68]).

5.4.5. PILOT OPERATING SYSTEM:

Pilot is a personal computer operating system mostly written in Mesa Language. Mesa language provides monitors as the sole synchronizing tool.

In Mesa, the simplest monitor is an instance of a module, which is the basic unit of global programming structure. A Mesa module consists of a collection of procedures and their global data. Such a module has PUBLIC procedures which constitute the external interface and PRIVATE procedures whcih are internal to the implementation and cannot be called from outside the module; its data is normally entirely private. A MONITOR module differs only slightly. It has three kinds of procedures: entry, internal(PRIVATE) and external (non-monitor procedures). The first two are monitor procedures and execute with monitor lock held. The external procedure executes without monitor lock.

For a good description reader can refer to [LAM80].

5.5. Semaphores In IITKIX

The semaphore system calls in our implementation are vary similar in nature to that of Xenix-3 system calls. The difference is we have provided general semaphores rather than binary semaphores. Some more options like changing the value of the semaphore etc. are also added in our implemen-

tation.

We have avoided the generalization of Unix System V semaphore implementation due to the following reasons:

(a) Ease of use

(b) Need for the extra complexity was felt unnecessary.

Brief descriptions of the system calls are given below. For detailed description look into Appendix II.

5.5.1. Crsem:

```
sid = crsem(path,initval,mode);
```

Creat a semaphore by name *path* with the initial value *initval* and permission mode *mode*. Returns a semaphore identifier which can be used in semop and closem system calls.

5.5.2. Opensem

```
sid = opensem(path,oflag,initval,mode)
```

Open an existing semaphore specified by *path* or creat a new semaphore by name *path*.

5.5.3. Semop

```
rval = semop(sid,cmd,newval);
```

Do a semaphore operation specified by *cmd*. The following operations can be performed: PSEM (P operation), VSEM (V opertion),NDPSEM(No delay P operation),SETVAL(set

the initial value to 'newval'),GETVAL(get the current value of semaphore).

5.5.4. Closem

rval = closem(sid)

Close the semaphore specified by the semaphore identifier 'sid'.

An example in the next page shows how to use these system calls. Some of the classical synchronization problems are presented in Appendix III.

```
      /* This program just illustrates the
       * use of semaphore system calls
       * A correct but pointless program
       */
#include "sem.h"
main()
{
  int fd,val;
  char *fn = "/usr/cs/gram/sc1";
  extern crsem(),semop(),closem();

  fd = crsem(fn,1,Ø755);
  if (fd == -1) printf("can't creat sem %s ",fn);
  else {
    printf("semaphore value %d ",semop(fd,GETVAL));
    if(semop(fd,PSEM) == -1) error("psem");
    printf("critical sectionØ);
    if(semop(fd,VSEM) == -1) error ("vsem");
    closem(fd);
      }
}
error(s)
char *s;
{
  printf("error in %s ",s);
  exit(1)
}
```

5.6. Implementation Details

In our implementation "semaphore" is a special kind of file which contains the initial value as its data.

Before explaining how each system call has been implemented it is worthwhile to have a look at the relevant data structures. We have looked at some of the existing data structures (e.g INODE TABLE) already (see chapter 2.). In the "INODEE" TABLE entry we have added one pointer which points to SEMTABLE entry (see below) , if that file is a semaphore file. Otherwise this pointer will be null.

Now we shall have a look at some of the data structures introduced solely for the purpose of semaphore implementation.

SEM TABLE: Every semaphore currently in use has an entry in the global semaphore table (SEM TABLE). Each entry in this table records the current value of the semaphore (initially it is equal to the user specified "initval" in crsem or opensem system calls, which is also recorded). Apart from this it contains pointers to the head and tail of the semaphore wait queue - the queue of processes which are waiting for this semaphore value to become positive. It also contains a pointer to the INODE TABLE for obtaining certain parameters from the "inode".

SEMAPHORE WAIT QUEUE STRUCTURES: These structures

essentially contain a process id and a pointer to the next structure in the wait queue. At the time of system initialization all these structures are circularly linked with a header. If a process wants to wait on a semaphore (due to P operation), it obtains a structure from this avail queue, properly initializes the entries and appends the structure to the end of the wait queue of the corresponding semaphore and sleeps. Once that proces is woken up, it releases the structure which will be appended to the head of the semaphore avail queue.

Now let us have a brief look at how each system call has been implemented in its most general form. Names in capitals within parentheses refer to functions in the Unix Kernel.

CRSEM: Creats a new semaphore. A new inode entry is sought ("IALLOC"), if available, initialize the fields to proper values ("MAKNODE"), the mode being specified by the user, with the initial value specified by the user being written into the file ("WRITEI").The "mode" field of the inode indicates that it is a semaphore file. The new entry is now entered in its parent directory ("WDIR"). The created semaphore remains open for further use until closed.

OPENSEM: First the semaphore pathname is matched with the corresponding node entry ('NAMEI') and the inode is brought into the core ('IGET') if not already there. The first unused file descriptor from the PER PROCESS OPEN FILE

TABLE is allotted to this file ('UFALLOC'). This entry points to the system open FILE TABLE ('FALLOC') each of whose entries have fields pointing to the inode table.If a semaphore table entry is already there in the core corresponding to this inode, return the file descriptor as the semaphore identifier (sid). Otherwise a SEMTABLE entry is obtained ('SEMALLOC') and its fields are properly initialized, as explained in CRSEM. 'sid' is returned to the user.

CLOSEM: This closes a given semaphore id (i.e. the file descriptor). Its connection to the file table is detached, and unless any other 'sid' points to this semaphore, the file table entry too, is removed ('CLOSEF;CLOSEI'). The inode reference count is decremented ('IPUT') and if that is equal to zero release the incore inode entry and simultaneously clean-up the SEMTABLE entry corresponding to this sempahore ('DELSEM').

SEMOP: Does a semaphore operation depending upon the command. All these operations first obtain the semaphore table (SEMTABLE) entry through the file pointer, inode pointer and semaphore pointer chain. Now Let us examine the implementation of each command separately.

PSEM: This command performs a P operation on the specified semaphore. If the semaphore value is positive just decrement the value and return. Otherwise, the process obtains a semaphore queue wait structure

('SEMQALLOC') and initializes the entry with its process id. This structure is appended to the tail of the semaphore wait queue ('WAITQTAIL'). Then the process sleeps at an interruptible priority ('SLEEP') which causes the switching of the process to take place. When it is woken up by a VSEM call it is returned to the user without decrementing the value.

NDPSEM: If the semaphore value is positive decrement it by 1 otherwise return with error (No delay P operation). Programmer can use this to avoid deadlock situations.

VSEM: This implements V operation on the specified semaphore. First it checks whether any process is waiting for this semaphore .If more than one proces is waiting wake up the first process in the semaphore wait queue ('WAKEQHEAD') and returns without incrementing the semaphore value. Otherwise, it increments the value of the semaphore by 1 and returns. If we observe carefully, this method avoids "starvation" or permanent blocking and thus leading to a fair FIFO scheduling strategy.

GETVAL: Gets the current value of the semaphore.

SETVAL: If the current value of the semaphore is equal to the current initial value then the user specified new value is set as the new initial value.

5.7. CONCLUSION:

The important features of the scheme presented are :

[1] It provides a clean, uniform and consistent interface to the user, allowing him to operate in a very similar manner to that of files. The naming scheme (i.e. using file names for semaphores) makes the scheme more elegant and simple, which is not the case in some other systems(e.g. UNIX System V ([BAC86])).

[2] This scheme extracts lot of information (eg. owner etc.) from the already existing data structures (e.g. INODE TABLE) and thus avoiding lot of redundancy, which again is not the case in System V.

[3] Since, this scheme returns a descriptor from the user's private space, there is no way to corrupt the global data structures like SEM TABLE which is possible in some other systems (e.g. Unix System V ([BAC86])).

[4] This scheme does not provide a separate name space. In Unix System V a separate file system name space is used. Since, our implementation uses the OPEN FILE DESCRIPTOR TABLE, it forces the user to have less number of open files than that of System V. Thus it leads to underutilization of per process resources.

However, this scheme , like many other systems discussed, has also not eliminated the inherent problems of semaphores (i.e) detecting programming errors which will lead to deadlocks .

6. CONCLUSION

The whole exercise of enhancing IITKIX proved to be quite useful in understanding the Unix Kernel. In particular this exercise gave us a good insight into the areas of process management, file system and other related routines. It also highlighted the problems involved in the extension of a large piece of software such as the Unix Kernel.

We have tested our implementation through a large number of programs. In particular, we have checked for the boundary conditions such as writing outside the address space of the data segment. We have presented two classical problems of synchronization in Appendix III.

A critique of the feature added has already been given in earlier chapters.

We shall summarize, some of the main points which we have discussed throughout this report.

(a) We think we have achieved our main goal of providing a simple set of mechanisms to the user for the sharing of data among processes and for synchronization.

(b) In order to present a uniform name space to the user, we have used the file system naming space for naming shared segments and semaphores.

(c) Initialization of data segments and their creation is being done outside the Kernel. Utilities have been provided for this purpose.

(d) We can improve the present implementation of data sharing by introducing multiple shared data segments per logical segment of the virtual space if the protection modes of the segments are same.

(e) We can extend the concept of data sharing to include various options like, different process sharing different portions of the same segment with different access rights. Of course, we should keep in mind the limitations of the hardware.

(f) We can enhance the sempahore implementation by providing some more features like a process obtaining an array of semaphores rather than a single semaphore at a time and doing a semaphore operation on a set of semaphores (e.g. Unix System V). Implementation of higher level primitives (e.g. monitors) can also be explored.

(g) Throughout the project our aim was to get a "working" code rather than an efficient one. So, we can improve efficiency of the code by replacing those portions, where a conservative approach has been taken to solve the problem.

APPENDIX - I

CREATION AND USE OF SHARED SEGMENTS.

Shared segments allow many processes to share data. In our implementation the idea of sharing involves two stages.

(a) Creating a shared segment.

(b) Usng the created shared segment.

We have provided utilities, which pave a way for the user to create shared data segments. We have given a method through which user can initialize the data segment.

User has to describe the data segment under a single structure construct (see below). User has to include this structure while using the segment through 'shmatt' system call (see shmatt(2)).

Before illustrating the creation procedure let us define few terms.

(a) Bounded segments: These segments essentially contain pointers (i.e. references to other elements in the segment). So, these segments have to be attached to the same virtual address by every processe while using the segment. Hence, these segments have to be created with a particulsr virtual address (known as *bound address).*

(b) Unbounded segments: These segments do not contain any

pointers(i.e. references). So these segments can be attached any where in the address space.

[1] Every data segment has to be described using a single data structure. Suppose a user wants to have an array of structures of type 'msg' (shown below) then the description of the segment looks as follows:

```
struct  xyz {
    struct msg {
        int a;
        struct msg *b;
    }mesg[5Ø];
};
```

Note that here the array of 'msg' structures have been enclosed under a single struct 'xyz'. Essentially this is the structure shared by many users. Let this structure be available in some file, say "st.h".

[2] Based on the requirement user has to create a .c file using the following guidelines.

[a] If the segment is unbounded then creat a .c file using the format shown in "unbound.c". The segment may or may not be initialized. If you want to initialize, put the initialization code in the appropriate place.

[b] If the segment is bounded , then there are two cases.

(i) If this bounded segment need not be initialized , then use the format shown in "bnoinit.c" .

(ii) If this bounded segment has to be initial-

ized, use the format shown in "binit.c"

For example , let us assume that we want to create a linked list of elements from the structure shown above. Since the structure contains pointers , the segment we get from this structure is a bounded segment. To create a linked list, we have to initialize the pointers. Thus we want to create a bounded segment with initialization. Hence, we have to use the format shown in the file "binit.c". Let the file created in this manner be "btest.c". This file is shown below.

[3] Now, use the following command to create a shared segment. Let the name of the shared segment be "bshar".

/iitk/shar/mkshare bshar btest.c Ø755 1ØØØØ 1

Ø755 is the access mode of the shared segment. 1ØØØØ(hex) is the virtual address (i.e. the bound address of the segment). "1" represents that this shared segment has got to be initialized (see mkshare(1)).

The created shared segment , "bshar", can be used in *shmatt* system call.

The shared segment, which is a file in our implementation, contains a header followed by the data. This header essentially contains a magic number (to indicate that this is a shared segment), size of the segment (which is computed during the process of creation), whether it is bounded or not and virtual address (if any). This header is appropri-

ately interpreted by the *shmatt* system call.The structure of the header is available in /iitk/shar/header.h. (see sheader(5)).

Note: In the format files shown the user has to merely substitute the appropriate structure and initialization (if any) has to be done accordingly. He need not bother about the rest of the format file.

We have provided two system calls, *shmatt* and *shmdet*, for using the segments. *shmatt* attaches to the segment. *shmdet* detaches the segment. For details see shmatt(2),shmdet(2). For using the segments, user has to include the .h file in his program. He should declare a pointer of type 'xyz' (in the example shown above). The pointer is implicitly initialized while doing *shmatt* system call. For using the segment see the examples in Appendix III.

```
/*-------------------------------------------------*
 * FILE: unbound.c                                 *
 * This is the format file to creat unbound        *
 * shared segments                                 *
 *-------------------------------------------------*/

#include <data.h>
#include <fcntl.h>
#include "st1.h"   /* Replace "st1.h" by the appropriate
                      file name which contains the struct
                      to be shared */

struct xyz abc;    /* Throughout the program replace 'xyz'
                      by the appropriate structure name*/
main()
{
     int fd;

     fd = creat("/tmp/shr.o", Ø777);
     if(fd == -1)
           printf("can't creat /tmp/shr.o ");
     else {
         printf("created /tmp/shr.o");
         /*---------------------------------*
          * Put the initialization code     *
          * here, if any.                   *
          *---------------------------------*/

              abc.a = 2; /* A typical entry*/
                .
                .
                .
         /*---------------------------------*
          * Initialization ends here        *
          *---------------------------------*/

         write(fd,&abc,sizeof(struct xyz));
     }
}
/*<---------------------------------------------------->*/
/*-------------------------------------------------*
 * FILE: bnoinit..c                                *
 * This is the format file to creat bounded        *
 * shared segments, which are not initialized      *
 *-------------------------------------------------*/

#include <data.h>
#include <fcntl.h>
#include "st1.h"   /* Replace "st1.h" by the appropriate
                      file name which contains the struct
                      to be shared */

struct xyz abc;    /* Throughout the program replace 'xyz'
                      by the appropriate structure name*/
main()
{
     int fd;
```

```
    fd = creat("/tmp/shr.o", Ø777);
    if(fd == -1)
         printf("can't creat /tmp/shr.o ");
    else {
        printf("created /tmp/shr.o");
        write(fd,&abc,sizeof(struct xyz));
    }
}
/*<---------------------------------------------->*/
/*-----------------------------------------------*
 *  FILE: binit.c                                *
 * This is the format file for creating a bounded *
 * shared segment with initialization.           *
 *-----------------------------------------------*/
#include <data.h>
#include <fcntl.h>
#include "st.h"  /*st.h is the file which contains
                   the structure to be shared.
                   Put the appropriate file name
                   here instead of "st.h" */
                  /* replace 'xyz' by the appropriate
                    structure throughout the program
                    (i.e the structure specifed in
                    the file   "st.h" ). */
main(argc,argv)
int argc;
char *argv[];
{
    struct xyz *p;
    int vaddr,i,fd1,fd2,size,val;

    sscanf(argv[1],"%d",&vaddr);
    printf("bound %d",vaddr);
    size = sizeof(struct xyz);
    fd1 = creat("/tmp/shr.o",Ø755);
    if(fd1 == -1) error ("creat: /tmp/share");
    printf("created /tmp/shr.o");
    crhead(vaddr,size);
    if((fd2 = open("/tmp/head",O_RDWR)) == -1)
                 error("open:/tmp/head");
    p = (struct xyz *)shmatt("/tmp/head",vaddr,DWRITE);

       /*---------------------------------*
        * Put the initialization code here  *
        *---------------------------------*/
                 p->a = 3 ; /*a typical entry*/
                            /*Note: p is a pointer
                              to the struct 'xyz' */
```

```
        /*----------------------------------*
         * initialization ends here         *
         *----------------------------------*/

    val = write(fd1,(char *)p,size);
    if(val == -1) error("write");
    exit(Ø);
}

crhead(vaddr,size)
{
    int mn,bound,i,fd;

    mn = Ø412;
    bound = 1;

    fd = creat("/tmp/head",Ø755);
    if(fd == -1)error("creat /tmp/head");
    write(fd,(char *)&mn,4);
    write(fd,(char *)&size,4);
    write(fd,(char *)&bound,4);
    write(fd,(char *)&vaddr,4);
    for(i=Ø;i<size+16;i++)
       write(fd,"Ø",1);
}
error(s)
char *s;
{
  printf("error in %s  ",s);
  exit(1);
}

}
/*<---------------------------------------->*/

/*-----------------------------------------*
 * FILE: btest.c                           *
 * This is the test file for creating a linked   *
 * list of elements.                       *
 * The structure in "st2.h" has been shown       *
 * in a previous page.                     *
 * (This file is similar to binit.c)       *
 *-----------------------------------------*/

#include  <data.h>
#include <fcntl.h>
#include "st2.h"  /*st2.h is the file which contains
                    the structure to be shared*/

main(argc,argv)
int argc;
char *argv[];
{
    struct xyz *p;
    struct msg *m;
    int vaddr,i,fd1,fd2,size,val;
```

```
    sscanf(argv[1],"%x",&vaddr);
    printf("vaddr %x",vaddr);
    size = sizeof(struct xyz);

    fd1 = creat("/tmp/shr.o",Ø755);
    if(fd1 == -1) error("creat: /tmp/share");
    printf("created /tmp/shr.o");
    crhead(vaddr,size);
    if((fd2 = open("/tmp/head",O_RDWR)) == -1)
            error("open:/tmp/head");
    p = (struct xyz *)shmatt("/tmp/head",vaddr,DWRITE);

                /*---------------------------------*
                 * Put the initialization code here *
                 *---------------------------------*/

   for (m =(struct msg *)&p->mesg[Ø];
                     m<(struct msg *)&p->mesg[49];m++)
                m->b = m+1;
   m->b = (struct msg *)&p->mesg[Ø];

                /*---------------------------------*
                 * Initialization ends here         *
                 *---------------------------------*/
   val = write(fd1,(char *)p,size);
   if(val == -1) error("write");
   exit(Ø);
}

crhead(vaddr,size)
{
    int mn,bound,i,fd;

    mn = Ø412;
    bound = 1;

    fd = creat("/tmp/head",Ø755);
    if(fd == -1)error("creat /tmp/head");
    write(fd,(char *)&mn,4);
    write(fd,(char *)&size,4);
    write(fd,(char *)&bound,4);
    write(fd,(char *)&vaddr,4);
    for(i=Ø;i<size+16;i++)
       write(fd,"Ø",1);
}
error(s)
char *s;
{
  printf("error in %s  ",s);
  exit(1);
}
```

APPENDIX II
UTILITIES,SYSTEM CALLS AND FILE FORMATS

MKSHARE(1)

NAME

mkshare - create a shared segment

SYNOPSIS

/iitk/shar/mkshare sharfile formatfile mode [vaddr] [init]

DESCRIPTION

This program enables the user to create a data segment which can be shared by many user processes. User has to describe the data segment under a single data structure (see /iitk/shar/help). A shared segment has to be created before using it. *shmatt* system call allows the user to share the segment with many processes (see shmatt(2)).

The argument *sharfile* is the name of the shared segment , the user wishes to be created.. The *formatfile* parameter has to be a .c file. Depending upon the requirement user has to create this .c file based on one of the format files suggested (see /iitk/shar/help). *mode* (specified in octal) specifies the creation mode of the segment. The parameters *vaddr* and *init* are not required if the segment is unbounded (see /iitk/shar/help). *vaddr* indicates the virtual address with which the segment has to be created (known as *bound address* see /iitk/shar/help). This parameter has to be a hexadecimal number (see the impact of this address in shmatt(2)). This virtual address is of 32 bits. (note: The virtual address is forced to lie on a page boundary i.e (vaddr & ØxFFFFFØØØ) is the virtual address with which the segment is created.) The parameter *init* is necessary only if the user wants to have the bounded segment initialized. This parameter has to be 1 if the user wants to initialize a bounded segment.

EXAMPLE

/iitk/shar/mkshare bshar1 test1.c Ø755 1ØØØØ 1

This command creats a bounded, initialized shared segment "bshar1" with the bounded virtual address as 1ØØØØ(hex) and with the access mode 755(octal).

FILES

/iitk/shar/mkshare - This command file.

/iitk/shar/binit..c - Format file for a bounded ,initialized shared segment.

/iitk/shar/bnoinit.c - Format file for a bounded segment without initialization.

/iitk/shar/unbound.c - Format file for an unbounded

segment

/iitk/shar/help - help file: creation and using of shared segments.

SEE ALSO

shmatt(2),shmdet(2)

BUGS

Shell syntax may have unexpected effects for improper arguments.

NAME

shmatt - attach to the shared segment.

SYNOPSIS

```
#include <data.h>

shmatt(path,shmaddr,shmflg)

char *path;

int shmaddr,shmflg;
```

DESCRIPTION

shmatt attaches the shared segment specified by **path** to the data region of the calling process. The segment is attached at the address specified by one of the following criteria:

If the segment is not bounded (see /iitk/shar/help) then

(a) if **shmaddr** is equal to zero then the segment is attached to the virtual address selected by the system.

(b) if **shmaddr** is not equal to zero then the segment is attached to the address given by (**shmaddr** & OFFMASK). The constant OFFMASK is defined in <data.h> and is equal to ØxFFFFFØØØ.

If the segment is bounded then **shmaddr** is ignored and the segment is attached to its bound address (see mkshare(1)).
The segment is attached for reading if **shmflg** is **DREAD** , otherwise it is attached for reading and writing if **shmflg** is **DWRITE**

shmatt will fail and not attach the segment if one or more of the following aretrue:

path is not a shared segment **[ENOSHAR]**.

The creation mode of the segment does not allow the process to do the operation indicated by **shmflg** (see mkshare(1)) **[EACCES]**.

The available data space is not large enough to accommodate the shared segment **[ENOMEM]**.

shmaddr is not equal to zero and the value of (**shmaddr** & OFFMASK) is an illegal address **[EINVAL]**

The number of shared memory segments attached to the calling process would exceed the system imposed limit **[EMFILE]**.

A component of the path prefix is not a directory **[ENOTDIR]**.

A component of the path prefix does not exist **[ENOENT]**.

Search permission is denied on a component of the path prefix **[EACCES]**.

The path name is null **[ENOENT]**.

The file does not exist **[ENOENT]**.

path points outside the process's allocated address space **[EFAULT]**

A shared memory entry has to be created in the global table but the system imposed limit on the maximum number of shared memory segments allowed system wide would be exceeded **[ENOSPC]**.

RETURN VALUE

Upon successful completion, **shmatt** returns the data region start address of the attached segment. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

SEE ALSO

shmdet(2),mkshare(1).

ASSEMBLER

```
moveq #4Ø,dØ
movl  path,aØ
movl  shmaddr,d1
movl  shmflg,a1
trap  #Ø
```

NAME

shmdet - detach a shared segment.

SYNOPSIS

```
int shmdet(shmaddr)
int shmaddr
```

DESCRIPTION

shmdet detaches a shared memory segment attached using **shmatt** system call

shmdet will fail and not detach the shared memory segment if **shmaddr** is not the data region start address of a shared memory segment **[EINVAL]**.

RETURN VALUE

Upon successful completion **shmdet** returns a value of Ø . Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

SEE ALSO

shmatt(2),mkshare(1).

ASSEMBLER

```
moveq   #38,dØ
movl    shmaddr,a1
trap    #Ø
```

NAME

Crsem - creat a new semaphore .

SYNOPSIS

```
int crsem(path,initval,mode)
char *path;
int initval,mode;
```

DESCRIPTION

Crsem creats a new semaphore file by name **path** with initial value of the semaphore as **initval** and access mode as **mode** or returns an existing entry..

Upon successful completion, a non-negative integer, namely the semaphore identifier is returned, which can be used in semop,closem system calls.

Crsem will fail if one or more of the following is true

A component of the path prefix is not a directory **[ENOTDIR]**.

A component of the path prefix does not exist **[ENOENT]**.

Search permission is denied on a component of the path prefix **[EACCES]**.

The path name is null **[ENOENT]**.

The directory in which the file to be created does not permit writing **[EACCES]**.

The file exists but not a semaphore file **[ENOSEMFIL]**.

Twenty file descriptors are currently open **[EMFILE]**.

"path" points outside the process's allocated address space **[EFAULT]**.

A semaphore identifier is to be created but the system imposed limit on the maximum number of semaphores allowed system wide would be exceeded **[ENOSPC]**.

RETURN VALUE

Upon successful completion, a non-negative integer, namely a semaphore identifier is returned. Otherwise , a value of -1 is returned and **errno** is set to indicate the error.

SEE ALSO

intro(2),opensem(2),closem(2),semop(2)

ASSEMBLER

```
moveq   #49,dØ
movl    path,aØ
```

```
movl    initval,d1
movl    mode,a2
trap    #Ø
```

NAME

Opensem - open a semaphore.

SYNOPSIS

```
#include <sem.h>

int opensem(path,oflag,initval,mode)
char *path;
int oflag,initval,mode;
```

DESCRIPTION

Path points to the path name naming a semaphore. Opensem opens a file descriptor for the named file, which can be used in semop and closem system calls. Oflag values are constructed by or-ing flags from the following list.

SEM_OPEN: Open the semaphore. This flag should be present in all opensem system calls.

SEM_CREAT: This is or ed with SEM_OPEN and used. If the semaphore does not already exist, creat a new semaphore with initial value **initval** and permission mode **mode**.

SEM_EXCL: This is used in conjunction with SEM_CREAT Otherwise it has no effect. If SEM_EXCL and SEM_CREAT are set, opensem will fail if the file already exists.

Upon successful completion a non-negative integer, the semaphore id, is returned. The new file descriptor is set to remain open across exec system calls.

Opensem will fail if one or more of the following is true

A component of the path prefix is not a directory **[ENOTDIR]**.

A component of the path prefix does not exist **[ENOENT]**.

Search permission is denied on a component of the path prefix **[EACCES]**.

The path name is null **[ENOENT]**.

The semaphore file does not exist and the directory in which the file to be created does not permit writing **[EACCES]**.

Twenty file descriptors are currently open **[EMFILE]**.

"path" points outside the process's allocated address space **[EFAULT]**.

A semaphore identifier is to be created but the system

imposed limit on the maximum number of semaphores allowed system wide would be exceeded **[ENOSPC]**.

SEM_CREAT is not set and the named file does not exist **[ENOENT]**.

SEM_CREAT and SEM_EXCL are set, and the named file exists **[EEXIST]**.

File exists and it is not a semaphore file **[ENOSEM-FIL]**.

Oflag does not contain SEM_OPEN **[EINVAL]**.

RETURN VALUE

Upon successful completion, a non-negative integer, namely a semaphore identifier is returned. Otherwise , a value of -1 is returned and **errno** is set to indicate the error.

SEE ALSO

intro(2),opensem(2),closem(2),semop(2)

ASSEMBLER

```
moveq   #5Ø,dØ
movl    path,aØ
movl    oflag,d1
movl    initval,a1
movl    mode,d2
trap    #Ø
```

SEMOP(2)

NAME

semop - semaphore operations.

SYNOPSIS

```
#include &lt;sem.h>

int semop(sid,cmd,newval)
int sid,cmd;
int newval;
```

DESCRIPTION

Semop performs the specified operation indicated by **cmd** on the semaphore given by the semaphore identifier **sid.** User should have read permission to do any of these opertions.

The **cmd** has to be only one of the following.

PSEM: P opertion is performed. If the semaphore value is Ø the calling process sleeps ,until it is woken up by a V operation ,at an interruptible priority (and so it also wakes up on a receipt of a signal).Otherwise it decrements the value of the semaphore by 1. Returns Ø on success.

NDPSEM: This is similar to PSEM in that it decrements the value of the semaphore if it is positive. Otherwise it returns with error (No delay P opertion).

VSEM: V opertion is performed. It increments the value of the semaphore by 1,if no process is waiting for this semaphore. Otherwise it wakes up the first waiting process.

SETVAL: Change the initial value of the semaphore to **newval.** This operation is performed only if the current value of the semaphore is equal to the current initial value. Only the owner of the semaphore or superuser can use this **cmd** . Returns the old initial value on success.

GETVAL: Return the current value of the semaphore.

Semop fails if one or more of the following is true:

sid is not a valid file descriptor or file has not been opened with read access [EBADF].

sid is not a crsem or opensem returned semaphore identifier [ENOSEMFD].

cmd is not a valid command [EINVAL].

cmd is SETVAL and caller is neither the owner nor the super user [EPERM].

cmd is PSEM and no semaphore wait queue buffer is available for this buffer to wait in the semaphore wait queue **[ENOSPC]**.

cmd is NDPSEM and the value of the semahore is Ø **[EZERO]**.

cmd is VSEM and the resulting value exceeds the initial value **[ERANGE]**.

cmd is SETVAL and semaphore is in use **[EINUSE]**.

RETURN VALUE

Upon successful completion GETVAL returns the current value of the semaphore and SETVAL returns the old initial value. All other commands return Ø on success. Otherwise -1 is returned and **errno** is set to indicate the error.

SEE ALSO

crsem(2),opensem(2).

ASSEMBLER

```
        moveq   #56,dØ
        movl    sid,a1
        movl    cmd,d1
        movl    newval,a2
        trap    #Ø
```

CLOSEM(2)

NAME:

Closem - closem a semaphore identifier.

SYNOPSIS

```
int closem(sid)
int sid;
```

DESCRIPTION

sid is a semaphore identifier obtained from crsem or opensem system calls. Closem closes the file descriptor indicated by 'sid'. A close is automatic on all files on exit, but since there is a 20 open file limit on the number of open files per process, close is necessary for a procedure which will deal with many files.

Closem will fail if

[1] **sid** is not a valid open file descriptor **[EBADF]**.

[2] **sid** is not a crsem or opensem returned semaphore identifier **[ENOSEMFD]**.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

SEE ALSO crsem(2),opensem(2)

ASSEMBLER

```
moveq   #58,d0
movl    sid,a0
trap    #0
```

Carry bit set on failure and cleared on success.

NAME

sheader - header format of the shared segments.

DESCRIPTION

Every segment created using *mkshare* (see mkshare(1)) contains a header and then followed by the initialized data (if any). Layout of the header is :

```
/*
 * Layout of shared segment header file:
 * header of 32 bytes:
 *      magic number  412 ; /* 4 bytes*/
 *      segment size ;      /* 4 bytes*/
 *      bound infor. ;      /* 4 bytes*/
 *      virtual address;    /* 4 bytes*/
 *      /*16 bytes unused*/
 *
 * header:        Ø
 * data  :        32  /* start of the data segment*/
 */
```

Note: This is very similar in nature to that of a.out header (see a.out(5)). This is to allow for possible future expansions.

APPENDIX - III

CLASSICAL PROBLEMS OF SYSNCHRONIZATION

In this appendix we shall present two classical problems of synchronization ,namely PRODUCER/CONSUMER problem, READERS-WRITERS problem.

1.PRODUCER/CONSUMER PROBLEM:

Producer/Consumer problems are quite common in operating systems. A *Producer* process produces information that is to be consumed by a *consumer* process. For example, a line printer driver produces characters which are consumed by the line printer.

In order to allow these processes to run concurrently, we must create a pool of buffers that can be filled and emptied by the producer and consumer, respectively. The producer and consumer must be synchronized ,so that the consumer does not try to consume items which have yet not been produced. In this situation the consumer must wait until a item is produced.

The *bounded-buffer* producer/consumer problem assumes that there is a fixed number of buffers. In this case, if all buffers are full the producer may have to wait.

We shall show the implementaion of *bounded-buffer* producer/consumer problem using a circularly linked list of 5Ø buffers. We have implemented this list in a shared seg-

ment (the creation of this has already been explained. see: Appendix I).

We have introduced three semaphores, namely, *full* ,which indicates the number of full buffers, *empty* ,which indicates the number of empty buffers, and *buf* which is used a 'Mutual exclusion' semaphore to avoid both consumer and producer simultaneously accessing the buffers. We have introduced one more semaphore, *shm* , which makes producer to wait till the consumer has got attached to the shared segment.

In the next few pages we show the implementation. We have omitted much of error checks for clarity purposes.

```
    /*-----------------------------------*
     *  FILE: st2.h                      *
     * This file contains the structure  *
     * which is to be shared by the      *
     * producer and the consumer         *
     *-----------------------------------*/
struct xyz {
  struct msg {
      int a;
      struct msg *b;
  }mesg[5Ø];

};
/*<---------------------------------------------->*/
      /*-------------------------------------*
       * Producer  process                   *
       * The shared segment "bsharl" has     *
       * already been created as explained   *
       * in the previous pages. It contains  *
       * a circularly linked list of 'msg'   *
       * structures  shown above.            *
       *-------------------------------------*/
#include <data.h>
#include <sem.h>
#include "st2.h"
main()
{
   struct xyz *s;
   struct msg *p;

   int sid1,sid2,sid3,i,val,sid;
        /*
         * Open semaphore files
         */
   sid1 = opensem("empty",SEM_OPEN | SEM_CREAT,5Ø,Ø777);
   sid2 = opensem("full",SEM_OPEN | SEM_CREAT,Ø,Ø777);
   sid3 = opensem("buf",SEM_OPEN | SEM_CREAT,1,Ø777);
   sid = opensem("shm",SEM_OPEN | SEM_CREAT,Ø,Ø777);
   printf("sid1val %d ",semop(sid1,GETVAL));
   printf("sid2val %d ",semop(sid2,GETVAL));
   printf("sid3val %d ",semop(sid3,GETVAL));
   if(sid1 == -1 || sid2 == -1 || sid3 == -1 ||sid == -1)
             error ("open: sids");
        /*
         * Attach to the shared segment
         */
   s =(struct xyz *) shmatt("bsharl",Øx1ØØØØ,DWRITE);

        /*
         * wait for the consumer to attach
```

```
        */
   val = semop(sid,PSEM);
   p = (struct msg *)&s->mesg[Ø];
   i = Ø;
ploop:
      i += 1Ø;
        /*
         * Decrement the "empty" semaphore.
         */
      val = semop(sid1,PSEM);
        /*
         * Can I use the buffers ?
         * Wait if consumer is using it.
         */
      val = semop(sid3, PSEM);

        /*
         * Producer gets the buffer list.
         * Critical Section starts.
         */

      p->a = i;
      printf("produced item %d ",p->a);
      p = p->b;

        /*
         * Critical Section ends
         */

      val = semop(sid3, VSEM);
      val = semop(sid2, VSEM);
      goto ploop;
}
error(s)
char *s;
{
  printf("error in %s ",s);
  exit(1);
}

/*<-------------------------------------------------->*/

     /*-----------------------------------*
      * consumer process                  *
      * The shared segment "bshar1" has   *
      * already been created as explained *
      * in the previous pages. It contains *
      * a circularly linked list of 'msg' *
      * structures  shown above.          *
      *-----------------------------------*/

#include <data.h>
#include <sem.h>
#include "st2.h"

main()
```

```
{
   struct xyz *s;
   struct msg *p;

   int sid1,sid2,sid3,i,val,sid;
        /*
         * Open semaphore files
         */
   sid1 = opensem("empty",SEM_OPEN | SEM_CREAT,5Ø,Ø777);
   sid2 = opensem("full",SEM_OPEN | SEM_CREAT,Ø,Ø777);
   sid3 = opensem("buf",SEM_OPEN | SEM_CREAT,1,Ø777);
   sid = opensem("shm",SEM_OPEN | SEM_CREAT,Ø,Ø777);
   printf("sid1val %d ",semop(sid1,GETVAL));
   printf("sid2val %d ",semop(sid2,GETVAL));
   printf("sid3val %d ",semop(sid3,GETVAL));
   if(sid1 == -1 || sid2 == -1 || sid3 == -1 ||sid == -1)
             error ("open: sids");

        /*
         * Attach to the shared segment
         */
   s =(struct xyz *) shmatt("bshar1",Øx1ØØØØ,DWRITE);

        /*
         * Signal to the producer that
         * Consumer has attached to the segment.
         */
   val = semop(sid,VSEM);
   p = (struct msg *)&s->mesg[Ø];
cloop:
        /*
         * Decrement the "full" semaphore.
         */
      val = semop(sid1,VSEM);
        /*
         * Can I use the buffers ?
         * Wait if producer is using it.
         */
      val = semop(sid3, PSEM);

        /*
         * Producer gets the buffer list.
         * Critical Section starts.
         */
      printf("consumed item %d ",p->a);
      p = p->b;

        /*
         * Critical Section ends
         */
      val = semop(sid3, VSEM);

        /*
```

```
        * Increment the "empty" semaphore.
        */
      val = semop(sid1, VSEM);
      goto cloop;
}
error(s)
char *s;
{
  printf("error in %s ",s);
  exit(1);
}
```

2. READERS/WRITERS PROBLEM:

A data object may have to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object (known as *readers*), while others may want to updte the shared object (known as *writers*).

Many readers can access the shared object simultaneously, while no more than one writer can access the object at the same time. Also while a writer is using the object, no reader can access it. When both readers and writers are waiting to access the object, we have two choices:

(a) Give priority to readers(known as *first* readers/writers problem).

(b) Give priority to writers (known as *second* readers/writers problem).

We have implemented the second readers/writers problem in the following m manner. We maintain three queues.

(1) Avail queue: Circularly linked list of free buffers

(2) Readers queue: List of waiting readers.

(3) Writers queue: List of waiting writers.

Whenever a process wishes to wait it gets a free buffer from the avail queue, initializes the entries and attaches to the particular queue. We are using three common variables:

(a) wc : Number of processes either writing or waiting to

write.

(b) rc: Number of processes reading at the moment.

busy: A boolean variable- to indicate write is in progress.

Apart from these variables every process has a private semaphore which is initialized to Ø. Any process can do a V operation on that semaphore. But only the owner process can do a P operation on that semaphore. We have put all these variables in a shared segment so that it can be shared by all reders and writers.

In the following pages we show the implementation.

```
/*---------------------------------------*
 * FILE: queue.h                         *
 * This file contains the structure      *
 * of the shared memory segment.         *
 * for controlling access to the shared  *
 * object. Note: This segment has nothing *
 * to do with the shared segment which   *
 * readers and writers wish to access.   *
 * After initializing (initialization    *
 * has not been shown) the segment will  *
 * contain a circularly linked list of   *
 * queue structures  in the avail queue. *
 * The read and write counts will be zero.*
 * The flag 'busy' is set to false       *
 *---------------------------------------*/

#define TRUE  1
#define FALSE Ø
#define NULL Ø

      struct  queue  { /* queue structure */
         short pid;        /*process id */
         struct queue *q; /*pointer to the next element*/
      };
   struct quseg  { /*structure of the queue segment*/

      struct queue qbuf[2Ø]; /*queue buffers*/
      struct queue availq  ; /*Avail queue header*/
      struct queue readq   ; /*read  queue header*/
      struct queue writeq  ; /*write queue header*/
      int rc;  /* readers count*/
      int wc;  /* writers count*/
      int busy; /*whether a process is writing ? */
   };
     /*------------------------------------------*
      * FILE rdwri.h                             *
      * The segment created  using this structure *
      * ("rwseg") is the one which is shared by  *
      * readers and writers.                     *
      * we have taken a simple data structure    *
      * just for illustration purposes.          *
      *------------------------------------------*/

      struct rw {
           int d[1ØØ];
      };
    /*----------------------------------------------*
     * This program shows the general structure
     * of a reader process
     * "rwseg" is the segment which is to be shared by
     * many processes for reading and writing.
     * The corresponding structure is shown in "rdwri.h"
     * "queseg" is the shared segment which contains
     * common variables (like availq, reaeq, etc.)
     * which are used for concurrency control.
     * (creation of these segments are  not shown).
```

```
    */
#include  <data.h>
#include  <sem.h>
#include "rdwri.h" /* structure of the shared object*/
#include "queue.h" /* Structure for the common variable
                                          shared segment*/
main()
{
     int pid,siq,prsem,i;
     struct quseg *qseg;
     struct rw *rws;
     extern char *procsem();
     char *s;
        /*
         * Attach to the "queseg" shared segment
         * for accessing the common state variables
         */
     qseg = (struct quseg *)shmatt("queseg",0x10000,DWRITE);

        /*
         * Attach to the shared segment on which read
         * has to be done
         */

     rws = (struct rw *)shmatt("rwseg",0x0,DREAD);
     pid = getpid();
     s = procsem(pid);

       •/*
         * create a private semaphore
         */

     prsem = crsem(s,0,0755);

        /*
         * Semaphore for accessing the
         * common state variables
         */

     siq  = crsem("qsem",1,0755);
     start_read(qseg,pid,siq,prsem);

        /*
         * start reading
         */
     printf("reader %d is reading",pid);

     for (i=0;i <100;i++)
         printf("%d ",rws->d[i]);

       /*
        * reading over
        */

     end_read(qseg,prsem,siq);
 }
```

```
/*-------------------------------------------------------*
 * This program shows the general structure
 * of a writer process
 * "rwseg" is the segment which is to be shared by
 * many processes for reading and writing.
 * The corresponding structure is shown in "rdwri.h"
 * "queseg" is the shared segment which contains
 * common variables (like availq, reaeq, etc.)
 * which are used for concurrency control.
 * (creation of these segments are  not shown).
 */
#include  <data.h>
#include  <sem.h>
#include "rdwri.h" /* structure of the shared object*/
#include "queue.h" /* Structure for the common variable
                                    shared segment*/
main()
{
    int pid,siq,prsem,i;
    struct quseg *qseg;
    struct queue *qst;
    struct rw *rws;
    extern char *procsem();
    char *s;
       /*
        * Attach to the "queseg" shared segment
        * for accessing the common state variables
        */
    qseg = (struct quseg *)shmatt("queseg",Øx1ØØØØ,DWRITE);

       /*
        * Attach to the shared segment on which  write
        * has to be done
        */

    rws = (struct rw *)shmatt("rwseg",ØxØ,DREAD);
    pid = getpid();
    s = procsem(pid);

       /*
        * create a private semaphore
        */

    prsem = crsem(s,Ø,Ø755);

       /*
        * Semaphore for accessing the
        * common state variables
        */

    siq  = crsem("qsem",1,Ø755);
    start_write(qseg,prsem,siq);

       /*
        * start writing
```

```
        */
    printf("writer %d is writing",pid);
    for (i=Ø;i <1ØØ;i++)
        rws->d[i] = i+1ØØ;
      /*
       * writing over
       */

    end_read(qseg,prsem,siq);
}
    /*----------------------------------------*
     * FILE: rdwri.c                          *
     * This file contains the routines used by *
     * reader and writer processes.           *
     *----------------------------------------*/

#include <data.h>
#include <sem.h>
#include "queue.h"
extern char *procsem();
    /*-----------------------------------------
     * This routine has to be called by every
     * reader process before start reading
     * the shared object
     */

start_read(qseg,pid,siq,prsem)
struct quseg *qseg; /* Pointer to the queue segment*/
short pid;          /* The prcocess id of the calling
                                          process  */
int siq;     /* The semaphore id of queue semaphore*/
int prsem;     /* The semaphore id of process semaphore*/
{
     struct queue *qst;
     semop(siq,PSEM); /*Enter queue segment*/
     if (qseg->wc == Ø) init_read(qseg,prsem);
                          /* see below*/
     else {
               /*Get a free queue structure*/
         qst = (struct queue *)remq(&qseg->availq);
         qst->pid = pid; /*initialize*/
         addq(&qseg->readq,qst); /*Wait in the readq*/
     }
     semop(siq,VSEM); /*Allow others to access the
                          queue segment */
     semop(prsem,VSEM); /*reader can proceed*/
}
    /*----------------------------------------
     * Activate a given reader process
     */

init_read(qseg,prsem)
struct quseg *qseg;
```

```
{
     qseg->rc++;
     semop(prsem,VSEM);
}

    /*-------------------------------------
     * Every writer process has to call the
     * following routine before writing into
     * the shared object
     * Most of the comments shown above hold
     * good for this routine, as well as the
     * other routines to come
     */

start_write(qseg,pid,siq,prsem)
struct quseg *qseg;
short pid;
int prsem,siq;
{
     struct queue *qst;

     semop(siq,PSEM);
     qseg->wc++;
     if((qseg->rc == 0) && (!(qseg->busy)))
            init_write(qseg,prsem); /* see below*/
     else {
         qst = (struct queue *)remq(&qseg->availq);
         addq(&qseg->writeq,qst); /*wait in the writers Q*/
     }
     semop(siq,VSEM);
     semop(prsem,VSEM); /*writer can proceed*/
}

    /*-------------------------------------
     * Activate a given writer process
     */

init_write(qseg,prsem)
struct quseg *qseg;
{
    qseg->busy = TRUE;
    semop(prsem,VSEM);
}

    /*-------------------------------------
     * This routine has to be called by every
     * reader process after reading the shared
     * data object.
     */

end_read(qseg,prsem,siq)
struct quseg *qseg;
{
    struct queue *qst;
    int opsem;
    char *s;

    semop(siq,PSEM);
```

```
    qseg->rc--;
    /*
     * If any writer is waiting allow him to
     * proceed, if no reader is currently
     * using the segment
     */
    if((qseg->rc == 0) && !(empty(&qseg->writeq))) {
            /*
             * Get the first process from the write Q
             */
            qst = (struct queue *)remq(&qseg->writeq);
            s = procsem(qst->pid);
            opsem = opensem(s,SEM_OPEN);
            addq(&qseg->availq,qst);
            init_write(qseg,opsem);
    }
    semop(siq,VSEM);
}
    /*-------------------------------------------
     * This routine has to be called by every writer
     * process after writing into the shared object
     */
end_write(qseg,prsem,siq)
struct quseg *qseg;
{
     struct queue *qst;
     int opsem,rsem;
     char *s;
     semop(siq,PSEM);
     qseg->wc--;
     qseg->busy = FALSE;

        /*
         * If any writer is waiting
         * allow him to proceed before the
         * readers
         */
     if(!(empty(&qseg->writeq))) {

        /*
         * Get the first process from the write Q
         */
        qst = (struct queue *)remq(&qseg->writeq);
        s = (char *)procsem(qst->pid);
        opsem= opensem(s,SEM_OPEN);
        addq(&qseg->availq,qst);
        init_write(qseg,opsem);
     }
     else
       while (!(empty (&qseg->readq))) {
```

```
                /*
                 * Get the first process from the read Q
                 */
                qst = (struct queue *)remq(&qseg->readq);
                s = (char *)procsem(qst->pid);
                rsem = opensem(s,SEM_OPEN);
                addq(&qseg->availq,qst);
                init_read(qseg,rsem);
         }
      semop(siq,VSEM);
}
      /*-------------------------------------
       * This routine converts the process id into
       * a semaphore of the corresponding process.
       * (e.g. "proc32").
       */
char *procsem(pid)
short pid;
{
       char *s;

       sprintf(s,"proc%d",pid);
       return(s);
}
      /*---------------------------------*
       * FILE: queue.c                   *
       * This file contains the routines *
       * to manipulate the queues        *
       * Structures are shown in "queue.h" *
       *---------------------------------*/
#include "queue.h"

      /*
       * Add a queue structure "e" to the list
       * pointed by "qhead"
       */

addq(qhead,e)
struct queue *qhead,*e;
{
    while (qhead->q != qhead)
      qhead = qhead->q;

    e->q = qhead->q;
    qhead->q = e;
}
      /*-------------------------------
       * Return the first queue structure
       * from the list pointed by "qhead"
       */

struct queue *remq(qhead)
struct queue *qhead;
{
    struct queue *save;
```

```
    if ((save = qhead->q) != qhead) {
        qhead->q = save->q;
        save->q = NULL;
        return(save);
    }
    return(NULL);
}
     /*------------------------------
      * Check whether the queue is empty
      */
empty(qhead)
struct queue *qhead;
{
    if (qhead->q == qhead)
       return(1);
    return(Ø);
}
```

REFERENCES

[ACC86]: Mike Accetta et al., *Mach: A new kernel foundation for UNIX development.* USENIX conference, Summer 1986,pp 93-112.

[AHU86]: Sudhir Ahuja et al., *Linda and friends.* Computer, August 1986,pp 26-34.

[APO85]: *Programming with system calls for interprocess communication.* Manual, Apollo Computer Inc.,1985.

[ARN86]: James Q. Arnold, *Shared libraries on Unix System V.* USENIX conference, Summer 1986,pp 395-404.

[BAC86]: Maurice J. Bach, *Design of the UNIX operating system.* Prentice Hall Inc.,N.J.,1986.

[BRI73]: Brinch Hansen.P, *Operating System principles,* Prentice Hall of India, 1973.

[CAR86]: Nicholas Carriero and David Gelernter, *The S/Net's Linda Kernel,* ACM Transactions On Computer Systems,4,2,May 1986,pp 110-129.

[COO85]: R.E.M.Cooper, *Porting Unix Quart to Data General Eclipse Pint,* Software Practice & Experience,vol.15,No.6,June 1985.

[DAL68]: Robert C. Daley and Jack B.Dennis, *Virtual Memory, Processes and Sharing in MULTICS.* CACM,11,pp 306-312,1968.

[DIJ68]: Dijkstra E.W., *The structure of THE multiprogramming system.* CACM,11,pp 341-346,1968.

[ELX84]: *ELXSI System 6400 introduction,* 3 ed,April 1984.

[HAM86]: Paul Hamdak, *Para functional Computing.* Computer, August 1986,pp 60-69.

[HWA87]: Kai Hwang et al., *Computer Architectures for AI processing.* Computer,January 1987,pp 19-27.

[KAL86]: Kalyan K. Banerjee, *Porting Unix V6 to S-32(II),* M.Tech Thesis Report, Dept. of C.S.E.,IIT Kanpur,January 1986.

[LAG78]: Klaus Lagally, *Synchronization in a Layered System.* Lecture Notes in Computer Science,Vol.60,1978,pp 253-280.

[LAM80]: Butler W.Lampson and David D.Redell, *Experience with Processes and Monitors in Mesa.* CACM,23,2,February 1980,pp 105-117.

[LIO77a]: J.Lions, *Unix Operating System source code level 6.* November 1977,Unpublished document.

[LIO77b]: J.Lions, *A commentary on the Unix Operating System.* November 1977,Unpublished document.

[LIS75]: A.M.Lister, *Fundamentals of Operating Systems,* Macmillan Computer Science Series,1975.

[LUN87]: Stephen F.Lundstorm, *Application considerations in the system design of highly concurrent Multiprocessors.* IEEE Transactions on Computers,November 1987.

[MOR87]: Nicholas Morkhaff, *Parallelism breeds a new class of Super Computers.* Computer Design,March 1987.

[PET]: James L. Peterson and Abraham Silberschatz, *Operating System Principles.* Addison Wesley,1983.

[QUA85]: J.S.Quarterman et al., *4.2 BSD and 4.3 BSD as examples of the Unix system.* ACM Computing Survey,December 1985.

[REE77]: David P. Reed, Rajendra K. Kanodia, *Synchronizations with eventcounts and sequencers.* Proceedings of the Sixth Symposium on O.S. Principles, ACM

[RIT74]: D.M.Ritchie and K. Thompson, *The UNIX time-sharing system.* CACM,17,7,July-August 1974,pp 365-375.

[ROC]: Marc.J.Rockhind, *Advanced UNIX Programming.* Prentice Hall Inc.,N.J.,1985.

[SAL78]: J.H.Saltzer, *Naming and Binding of Objects.* Lecture Notes in Computer Science,Vol.60,1978,pp 193-208.

[SHA74]: Alan C.Shaw, *The Logical Design of Operating Systems,* Prentice Hall Series in Automatic Computation,1974.

[SRI86]: Shri Kant Srivastava, *Porting Unix V6 to S-32(I).* M.Tech Thesis Report, Dept. of C.S.E.,IIT Kanpur,January 1986.

[SYSV]: *SYS V Unix Programmer Reference Manual.* PLEXUS Computers Inc.,1985.

[SZN86]: Edward W.Sznyter et al., *A new virtual memory implementation for Unix* USENIX conference, Summer 1986,pp 81-92.

[UNIXa]: *UNIX System Papers.* Manual, Horizon II/III,Version 1.0,1986.

[UNIXb]: *UNIX Programmer's Manual.* Part III,Horizon II/III, Version 1.0, 1986.

[UNOS]: *UNOS Subroutines.* Manual, Sunray Computers, 1984.